THM TECHNISCHE HOCHSCHULE MITTELHESSEN
CAMPUS GIESSEN
MNI Mathematik, Naturwissenschaften und Informatik

Modul II 1002 im Studiengang
BSc Ingenieur-Informatik

Mikroprozessortechnik

Prof. Dr. Klaus Wüst

Skriptum zum Mitdenken und Ergänzen

Stand: February 11, 2020
Autor: K.Wüst

Vorwort

Diese Skriptum soll Sie durch eine Veranstaltung führen, die aus Vorlesung, Übungen und Praktikum besteht. Die Übungen sind in das Skriptum integriert, meistens am Ende des Kapitels. Für das Praktikum gibt es eine eigene Anleitung. Wir versuchen, Vorlesung, Übungen und Praktikum aufeinander abzustimmen.

Für Hinweise auf Fehler sind wir jederzeit dankbar!

Viel Spaß bei der Mikroprozessortechnik!

Klaus Wüst

Contents

1 Einführung

1.1 Mikroprozessorsysteme

Alle Computer, mit denen wir heute arbeiten sind Mikroprozessorsysteme. Sie enthalten außer dem Mikroprozessor auch Programmspeicher, Datenspeicher und Ein-/Ausgabeeinheiten.

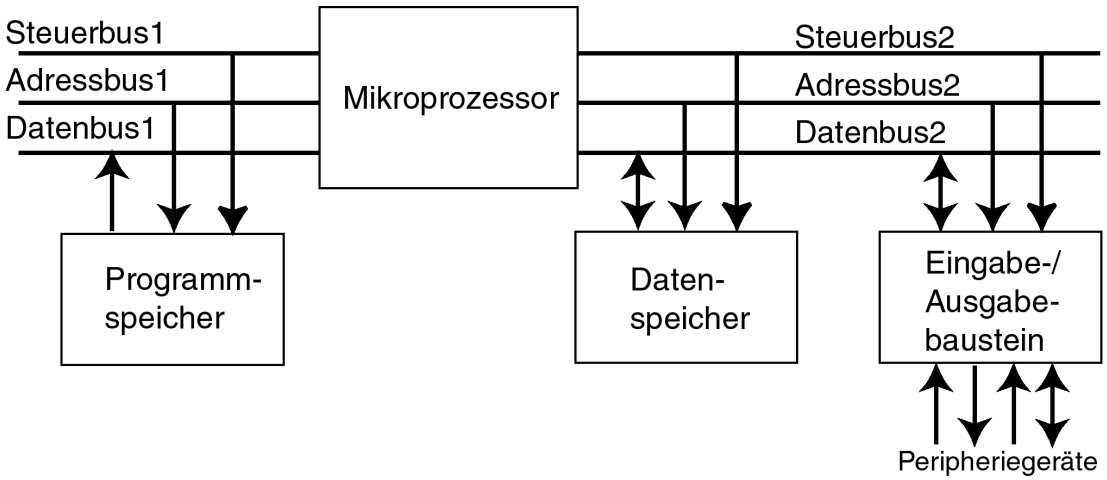


Figure 1.1: In einem Mikroprozessorsystem sind alle Bausteine an gemeinsame Leitungssträngen angeschlossen, die Bussysteme. Bei der Harvard-Architektur gibt es einen Programmspeicher und einen Datenspeicher.

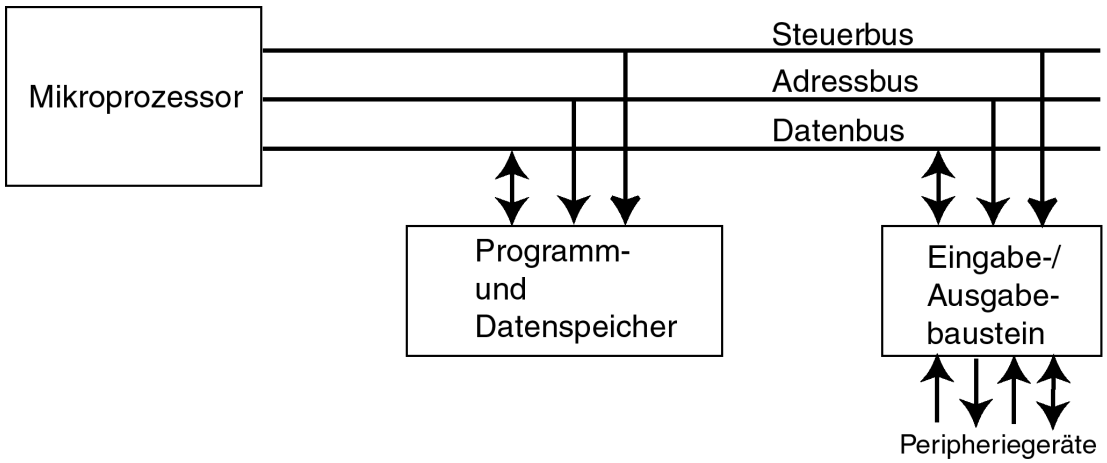


Figure 1.2: Bei der von-Neumann-Architektur befinden sich Programm und Daten in einem gemeinsamen Speicher

1.2 Aufbau eines Mikroprozessors

Aufgabe im System

Jedes Rechnersystem hat eine Central Processing Unit, (CPU, auch Zentraleinheit). Diese ist Kernstück und "'Master"' des Computers, steuert die Ausführung der Programme, übernimmt die eigentliche Verarbeitung der Daten und steuert alle Datentransfers über die Busse von und zu den Speichern und den Peripherieeinheiten.

Die CPU war früher eine eigene Baugruppe ("'Prozessorkarte"') und ist heute ein einziger integrierter Schaltkreis. (Integrated Circuit, IC, Chip) Dieser Chip heißt Mikroprozessoren ("'kleiner Prozessor"')

Es gibt auch Mikroprozessoren für spezielle Aufgaben, z.B. *Signalprozessoren, Mikrocontroller, Arithmetik-Prozessoren* und *Kryptographieprozessoren*.

Bestandteile eines Mikroprozessors

Alle Mikroprozessoren bestehen in ihrem Inneren aus mehreren Baugruppen, die für verschiedene Aufgaben zuständig sind (Abb. 1.3).

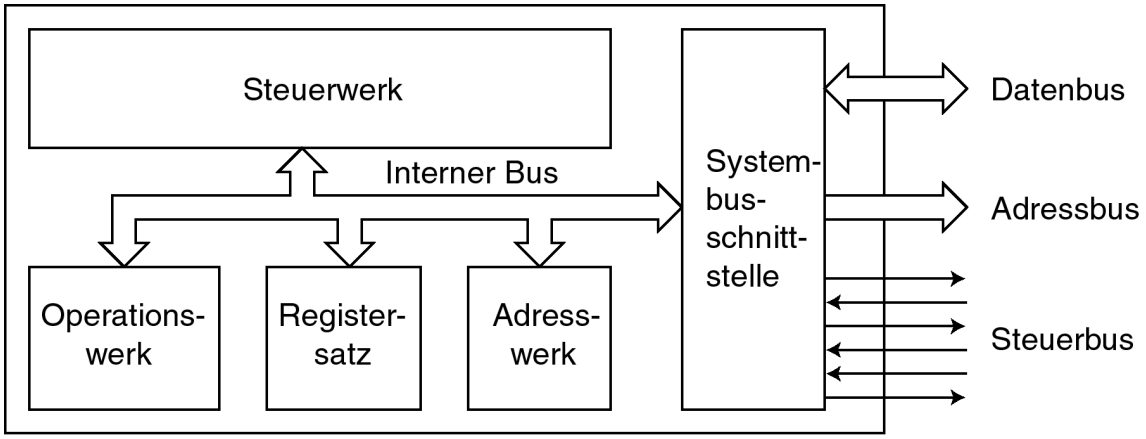


Figure 1.3: Interner Aufbau eines Mikroprozessors. (Blockschema)

Der Registersatz enthält einen Satz von Registern, mit dem Daten innerhalb des Prozessors gespeichert werden können. Ein Register ist eine Gruppe von Flipflops mit gemeinsamer Steuerung.

Das Operationswerk führt die eigentliche Verarbeitung, d.h. die logischen und arithmetischen Operationen, an den übergebenen Daten aus.

Das Steuerwerk ist verantwortlich für die Ablaufsteuerung sowohl im Inneren des Prozessors als auch im restlichen System.

Das Adresswerk erzeugt die erforderlichen Adressen, um auf Daten und Code im Hauptspeicher zugreifen zu können.

Die Systembus-Schnittstelle enthält Puffer- und Treiberschaltungen, um den Daten-

verkehr über den Systembus abzuwickeln.

Maschinenbefehle

Der Maschinenbefehlssatz beschreibt jede Aktion, die der Prozessor ausführen kann, als einen Maschinenbefehl. Typische Maschinenbefehle sind

- arithmetische Operationen
- bitweise logische Operationen
- Zugriffe auf Datenspeicher
- Zugriffe auf Ein-/Ausgabebausteine

Befehle und Daten

- Befehle sind binär codierte Bitmuster, die den Prozessor anweisen, bestimmte Operationen auszuführen.
- Daten sind binär codierte Bitmuster, bestimmter Länge (z.B. 8, 10, 12, 16 oder 32-Bit). Sie werden durch die Befehle auf bestimmte Art interpretiert, z.B. als vorzeichenlose 32-Bit-Zahl oder als 8-Bit-Zeichen

Busleitungen

Die Verbindung der Komponenten erfolgt durch elektrische Leiterbahnen und Leitungen. *Busleitungen* oder einfach *Busse* sind Leitungsbündel, an denen mehrere Komponenten parallel angeschlossen sind. Bei einer Übertragung transportiert jede Busleitung ein Bit, wobei die Einsen und Nullen der Bitmuster auf den Busleitungen als HIGH- und LOW-Pegel dargestellt werden (Abb. 1.4).

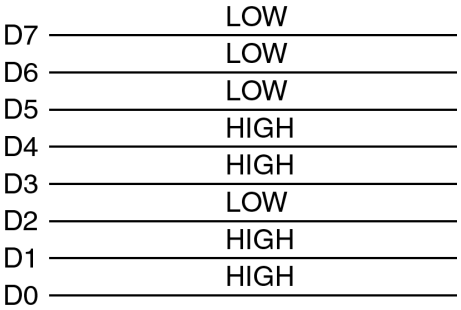


Figure 1.4:
Die Übertragung des Bitmusters 00011011b auf einem 8-Bit-Datenbus

Datenbus Leitungen, über die Befehle und Daten übertragen werden, besteht aus vielen parallelen, gleichartigen Datenleitungen. *Zugriff: schreiben und lesen*

Adressbus Leitungen, mit denen in Speicherbausteinen und Ein-/Ausgabebausteinen bestimmte Plätze angewählt werden, besteht aus vielen parallelen, gleichartigen Adressleitungen. Die übermittelten Bitmuster sind die Adressen. Mit n Adressleitungen können 2^n verschiedene Bitmuster dargestellt werden und somit 2^n Adressen angesprochen werden. *Zugriff: nur schreiben*

Steuerbus Leitungen, mit denen Bausteine in Ihrer Arbeitsweise gesteuert und koordiniert werden. Besteht aus vielen unterschiedlichen Steuerleitungen. Im

Gegensatz zum Datenbus und Adressbus hat im Steuerbus jede Leitung eine ganz spezielle Bedeutung. *Zugriff: unterschiedlich*

Verbindungen zur Außenwelt: Ein- und Ausgabe

- *Eingabebausteine* nehmen Signale von Peripheriebausteinen und externen Geräten, z.B. einer Tastatur, entgegen. Der Eingabebaustein legt sie dann auf den Datenbus auf dem sie an den Prozessor übermittelt werden.
- *Ausgabebausteine* dienen dazu, Signale an externe Geräte bzw. Peripheriegeräte auszugeben, z.B. an einen Grafik-Controller, an den ein Bildschirm angeschlossen ist. Dazu legt der Prozessor das gewünschte Bitmuster auf den Datenbus, von dort kommen sie auf den Ausgabebaustein, der es an die Peripherie weitergibt.

1.3 Arbeitsweise von Mikroprozessoren

Aufgaben einer CPU/eines Mikroprozessors

1. Die CPU steuert alle notwendigen Funktionseinheiten, wie Speicher, Ein-/ Ausgabeeinheiten, Schnittstellen usw.
2. Die CPU führt die eigentliche Datenverarbeitung durch, d.h. die Bearbeitung von Bitmustern mit arithmetischen und logischen Operationen.
3. Die CPU sichert das korrekte Voranschreiten des Systemzustandes (Name Prozessor von procedere = voranschreiten.)

Die CPU ist eine komplexe Digitalschaltung. Realisiert als integrierter Schaltkreis, früher je nach Generation mit Relais, Röhren Transistoren oder Logikgattern (NAND/NOR) Mikroprozessoren sind immer als Integrierte Schaltung aufgebaut. Ein Mikrocomputer entsteht, wenn Mikroprozessoren in ein System aus Speicher, Ein-/Ausgabeeinheiten, Schnittstellen und Peripherie eingebettet werden.

Arbeitstakt

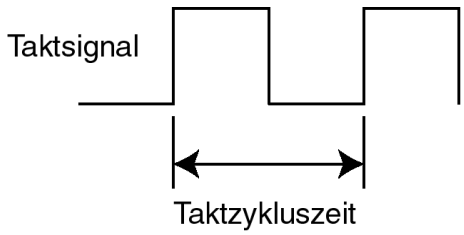


Figure 1.5: Taktsignal für einen Prozessor.

Jeder Mikroprozessor wird durch ein regelmäßiges Rechtecksignal synchronisiert, den Maschinentakt oder kurz Takt. Jeder Maschinenbefehl braucht eine bestimmte (bei RISC-Prozessoren von der Vorgeschichte abhängige) Anzahl an Taktzyklen. Die Dauer eines Taktzyklus T_c ist die reziproke Taktfrequenz f_A:

$$T_c = \frac{1}{f_A} \tag{1.1}$$

Jeder Befehl beansprucht eine bestimmte Anzahl von Taktzyklen, daraus kann mit der Taktfrequenz die Ausführungszeit des Befehles berechnet werden.

Beispiel Die Ausführungszeit eines Befehles mit 11 Taktzyklen auf einem System mit 2.5 MHz Prozessor-Taktfrequenz wird so berechnet:

Der Taktzyklus dauert $T_c = 1/f_A = 1/2500000\,\text{Hz} = 400\,\text{ns}$. Ein Befehl mit 11 Taktzyklen dauert also $400\,\text{ns} \cdot 11 = 4400\,\text{ns} = 4.4\,\mu\text{s}$. Bei vielen Befehlen hängt die Taktzahl allerdings von dem Typ der Operanden und weiteren Umständen ab.

Übung: 1.1 Prozessortakt

Wie schnell muss ein Prozessor getaktet werden, wenn eine Latenzzeit (Wartezeit) von 15 Takten nur 75 ns dauern darf?

Übung: 1.2 Ausführungszeit

Wie lange dauert die Ausführung einer Befehlssequenz mit insgesamt 15 Takten bei einem Prozessortakt von 2 GHz?

Ausführung des Maschinencodes

Jeder Prozessor verfügt über einen *Programmzähler* (Program Counter, PC), der die Adresse des nächsten auszuführenden Befehles enthält. Die grundsätzliche Funktionsweise eines Mikrorechnersystems ist nun eine endlose Wiederholung der folgenden Sequenz:

1. Auslesen des nächsten Befehls aus dem Programmspeicher, Programmzähler erhöhen.
2. Erkennen und Ausführen des Befehls; falls notwendig, werden auch Operan-

den aus dem Programmspeicher gelesen, auch dann muss der Programmzähler weitergerückt werden. Danach fortsetzen bei 1.

Sprünge werden einfach realisiert, indem der Programmzähler einen neuen Wert erhält. Damit werden Wiederholungsschleifen und Verzweigungen realisiert.

- Programmzähler wird mit einem kleineren Wert überschrieben → Rückwärtssprung → Code wird erneut ausgeführt → Schleife
- Programmzähler wird mit einem größeren Wert überschrieben → Vorwärtssprung → Code wird übersprungen → Bedingte Ausführung ("'if"')

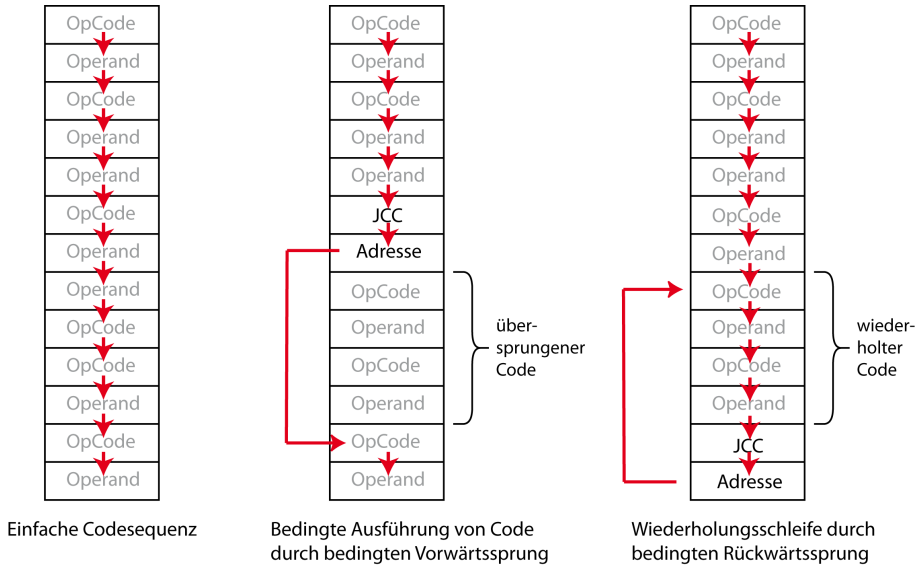


Figure 1.6: Die Grundstrukturen der Programmierung auf Maschinencode-Ebene..

Reset und Boot-Vorgang

- Der Reset ist der definierte Startvorgang, mit dem der Mikroprozessor seine Arbeit beginnt.
- Die Prozessorhardware garantiert eine vorgegebene Initialisierung der Register und Flags; dazu gehört auch der PC, das ergibt den definierten Einsprung in die Codeausführung
- Bei kleinen Systemen (Mikrocontrollern) startet nach dem Reset das Anwenderprogramm
- Bei größeren Systemen wird nach dem Reset der Urlader (Bootprogramm) gestartet, der das Betriebssystem lädt.
- Der unmittelbar nach dem Reset ausgeführte Code muss in einem nichtflüchtigen Speicher liegen

Als Quellen für die Auslösung des Reset-Vorgangs kommen in Frage:

- Externer Reset durch elektrisches Signal am RESET-Eingang,
- Reset beim Einschalten (Power On Reset),

- Reset bei Fehlerzuständen: Unterschreitung der zulässigen Betriebsspannung, unbekannter Opcode, fehlendes Rücksetzen der Watchdog-Schaltung und andere.

Übung: 1.3 Fragen zu Mikroprozessoren

1. Was sind die Hauptaufgaben einer CPU (=Zentraleinheit).
2. Nennen Sie mindestens zwei Mikroprozessoren für Spezialaufgaben.

1.4 Speicherung von Daten

Die kleinste Dateneinheit: Ein *Bit* (Abkürzung für *Binary digit*. Ein Bit kann die Werte 0 und 1 annehmen. Diese Werte werden technisch bzw. physikalisch auf unterschiedliche Art dargestellt:

- durch verschiedene Spannungspegel (Bus- und Schnittstellenleitungen),
- durch vorhandene oder nicht vorhandene leitende Verbindung (ROM),
- durch den Ladungszustand eines Kondensators (DRAM),
- durch den Zustand eines Flipflops (SRAM),
- durch den leitenden oder gesperrten Schaltzustand eines Transistors (Treiberbaustein),
- durch die Magnetisierungsrichtung eines Segmentes auf einer magnetisierbaren Schicht (magnetische Massenspeicher),
- durch die Reflexionseigenschaften einer spiegelnden Oberfläche (optischer Massenspeicher),
- durch den Polarisationszustand eines Ferroelektrikums (evtl. zukünftiger Speicherbaustein).

In Mikroprozessorsystemen werden fast immer mehrere Bit zu einer Informationseinheit zusammengefasst:

- 4 Bit sind eine *Tetrade* oder ein *Nibble*.
- 8 Bit sind ein *Byte*.

- Die Verarbeitungsbreite des Prozessors umfasst ein *Maschinenwort* oder *Wort*; bei einem Prozessor mit 32-Bit-Verarbeitungsbreite sind also 4 Byte ein Maschinenwort.
- Ausgehend vom Maschinenwort wird auch von *Halbworten, Doppelworten* und *Quadworten* (vier Maschinenworte) gesprochen; bei einem 16 Bit-Prozessor z.B. umfasst ein Quadwort 64-Bit.

Die meistgebrauchte Einheit: Das Byte, alle Speicher und Dateigrößen werden in Byte angegeben. Das niedrigstwertige Bit innerhalb eines Bytes oder Wortes heißt *Least Significant Bit* (LSB=Bit 0) das höchstwertige heißt *Most Significant Bit* (MSB).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
MSB							LSB

Für größere Informationseinheiten gibt es gebräuchliche Abkürzungen, die an die Einheitenvorsätze der Naturwissenschaften angelehnt sind:

2^{10} Byte	=	ein Kilobyte	=	1 KByte	=	1024 Byte	=	1024 Byte
2^{20} Byte	=	ein Megabyte	=	1 MByte	=	1024 KByte	=	1048576 Byte
2^{30} Byte	=	ein Gigabyte	=	1 GByte	=	1024 MByte	=	1073741824 Byte
2^{40} Byte	=	ein Terabyte	=	1 TByte	=	1024 GByte	=	1099511627776 Byte
2^{50} Byte	=	ein Petabyte	=	1 PByte	=	1024 TByte	=	1125899906842624 Byte
2^{60} Byte	=	ein Exabyte	=	1 EByte	=	1024 PByte	=	1152921504606846976 Byte

Oft werden diese Einheiten abgekürzt, z.B. "‘KByte"’ zu "‘KB"’, "‘MByte"’ zu "‘MB"’.

Der Stack

Der Stack (Stapel) ist ein besonderer Bereich des Speichers, der als LIFO-Speicher (Last In – First Out) verwaltet wird. Er dient zur vorübergehenden Aufnahme von Daten.

Die Verwaltung des Stack wird durch die Prozessorhardware unterstützt: Er besitzt ein spezielles Register, den *Stackpointer* (SP, Stapelzeiger).

Der Stack gleicht einem Stapel Teller, die immer nur einzeln aufgelegt werden. Nimmt man einen Teller von diesem Stapel, so ist es immer der Teller, der als letzter aufgelegt wurde. Traditionell wächst ein Stack immer zu den kleineren Speicheradressen hin. Man richtet es so ein, dass der Stackpointer immer auf das Wort an der Spitze des Stack (Top of Stack) zeigt. es gibt zwei Möglichkeiten:

- Der Stackpointer zeigt auf den ersten freien Platz
- Der Stackpointer zeigt auf den letzten belegten Platz (Abbildung)

Für den Stackzugriff gibt es traditionell die beiden Befehle Push und Pop, bei denen die Prozessorhardware auch den Stackpointer ändert:

Push legt ein neues Element an der Spitze des Stack ab und verkleinert den Stackpointer (vergrößert den Stack)

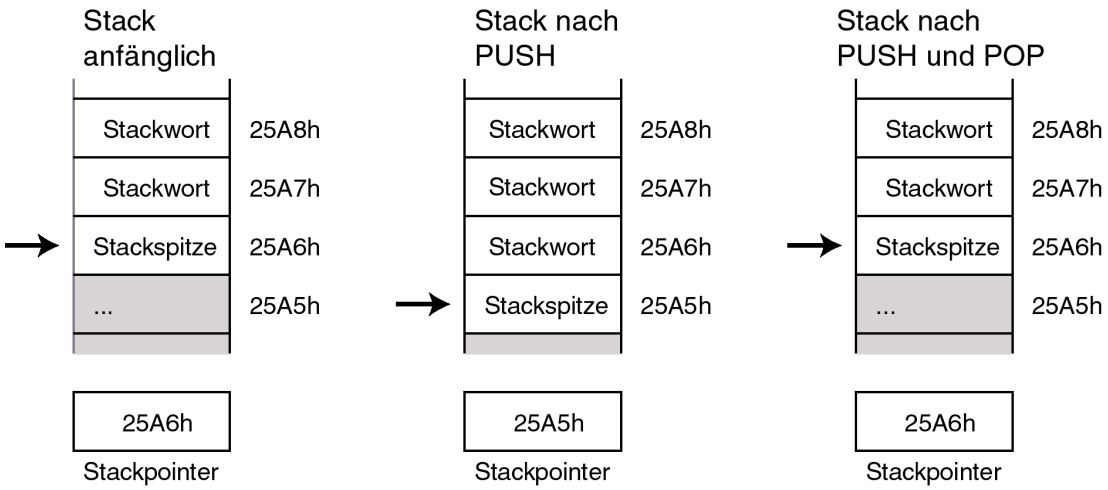


Figure 1.7: Ein Stack wächst abwärts. Mit den Befehlen PUSH und POP werden Daten auf dem Stack abgelegt bzw. vom Stack entnommen

Pop entnimmt das Element an der Spitze des Stack und vergrößert den Stackpointer (verkleinert den Stack)

Ergänzend: Es ist auch möglich den Stack register-indirekt zu adressieren und mit gewöhnlichen Transportbefehlen dort zu schreiben oder zu lesen; die Größe des Stack ändert sich dabei aber nicht.

Übung: 1.4 Frage zur Stackbenutzung

Wie kann man ausschließlich mit Stackbefehlen den Inhalt zweier Variablen A und B vertauschen?

2 Mikrocontroller

2.1 Allgemeine Eigenschaften

name: Mikrocontroller = "'kleine Steuerung"'. Bei einem Mikrocontroller (Abk. MC oder μC) sind die CPU, Speicher, Peripheriekomponenten und Interruptsystem auf einem Chip integriert. Ein Mikrocontroller kann also mit sehr wenigen externen Bausteinen betrieben werden, man nennt sie daher auch *Single-Chip-Computer* oder *Einchip-Computer*. Hier ist eine hohe *funktionelle Integration* wichtig: Je mehr Funktionen schon auf dem Mikrocontroller-Chip sind, um so weniger Zusatzbausteine braucht man.

> Ein Mikrocontroller enthält den Kern eines Mikroprozessors und zusätzlich Peripheriegruppen für Mess- Steuerungs- und Kommunikationsaufgaben.

Das Haupteinsatzgebiet der Mikrocontroller ist die Steuerung in *eingebetteten Systemen* (Embedded Systems).

> Ein eingebettetes System ist ein System, das von einen Computer gesteuert wird, ohne dass dieser nach außen in Erscheinung tritt.

Mikrocontroller werden in vielen (eingebetteten) Systemen eingesetzt, daher auch der Name: to control = steuern, Mikrocontroller = kleine Steuerung. Beispiele sind

- Kommunikationselektronik (Handys, schnurlose Telefone, Funkgeräte usw.)
- Automobile (Steuergeräte, Multimediaeinheiten, Wegfahrsperre)
- Unterhaltungselektronik (Radios, Fernseher, Bluray-Player ...)
- Geräte der Messtechnik und Medizintechnik (Röntgen, CT, Blutdruckmessgerät ...)
- Maschinen- und Anlagensteuerungen (Bearbeitungsmaschinen, Heizungen, Klimaanlagen, Solaranlagen ...)
- Haushaltsgeräte (Waschmaschinen, Mikrowellen, Herde ...)
- Haustechnik (Sprechanlagen, Rolladensteuerung, Alarmanlage ...)

Da heute die Systeme meistens eine Schnittstelle haben, um das Programm neu zu flashen, hat man viele Vorteile:

- höchste Flexibilität, da die Software die Funktion bestimmt
- Funktion änderbar
- Updates können eingespielt werden

- Fehlerbeseitigung möglich

Der Markt für Mikrocontroller wächst ständig. (2010: 16 Milliarden Systeme im Einsatz, 2020 erwartet: 40 Milliarden!, Umsatz: 9% Wachstum pro Jahr)

Mikrocontroller einer *Familie* haben gleiche oder ähnliche Kerne und unterscheiden sich in der Peripherie.

- Mikrocontroller-Familien können sehr groß sein (viele Peripheriekomponenten)
- Viele Hersteller produzieren mehrere MC-Familen
- es gibt viele MC-Hersteller
- Insgesamt: es gibt unglaublich viele Mikrocontroller!

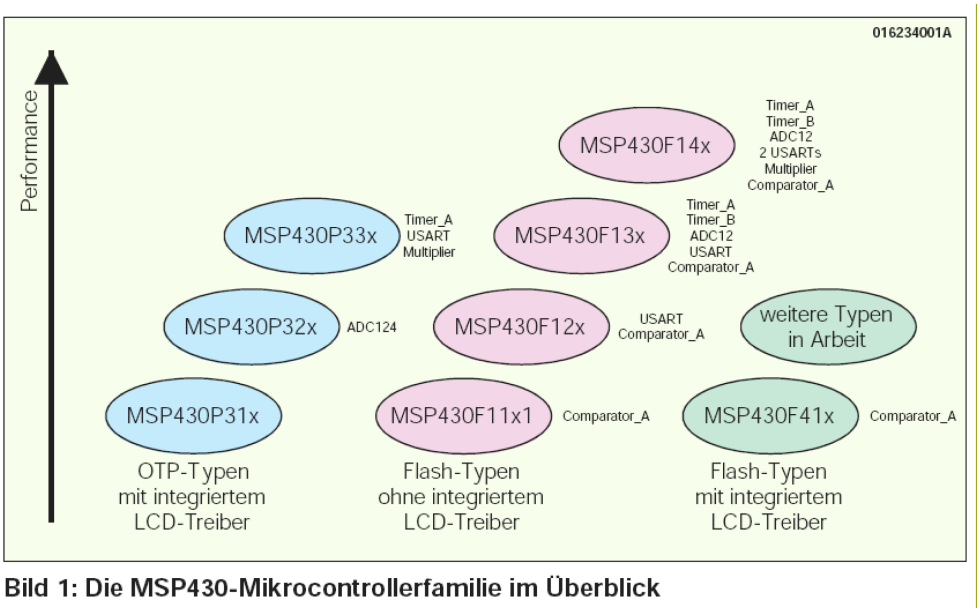


Bild 1: Die MSP430-Mikrocontrollerfamilie im Überblick

Figure 2.1: Famile der MSP430-Mikrocontroller (älterer Stand, Abb. mit freundlicher Genehmigung von Texas Instruments).

Der Kern (Core) der Mikrocontroller

- Er enthält Rechenwerk, Steuerwerk, Registersatz und Busschnittstelle; entspricht also ungefähr dem Kern eines Mikroprozessors,
- hat Register mit 4,8,16 oder 32 Bit,
- Prozessortakt 1 kHz - 100 MHz, dies bestimmt die Verarbeitungsgeschwindigkeit.

Peripheriegruppen der Mikrocontroller

(Auch Peripheriebausteine genannt) Übersicht:

- Verschiedene Arten von Speicher (RAM, Programm-Flash, Daten-Flash)

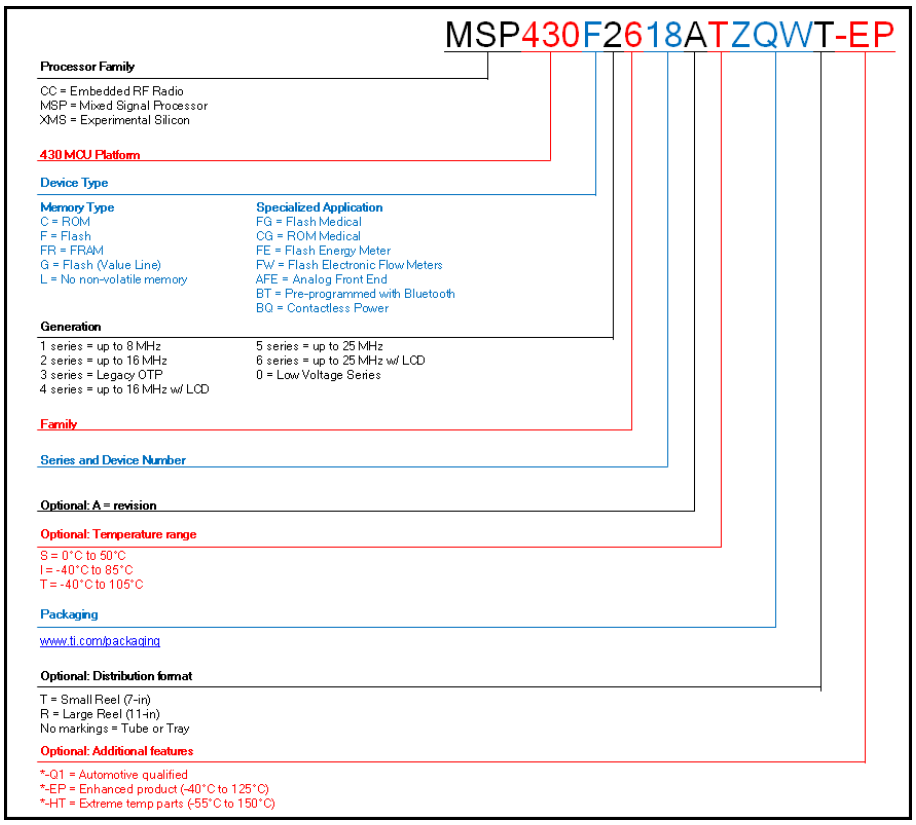


Figure 2.2: Schema der Namen für die MSP430-Familie. (Quelle: Wikipedia)

- Kommunikationsschnittstellen (UART, I^2C, SPI, CAN,...)
- Ein- und Ausgabeports (IO-Ports)
- Zähler/Zeitgeber-Bausteine
- Analog-Digital-Wandler
- Digital-Analog-Wandler
- Echtzeituhr (RTC)
- Ein konfigurierbares Interruptsystem mit externen Interrupts
- Watchdog Timer (WDT)
- Eine Oszillatorschaltung
- Ansteuerung von LCD- und LED-Anzeigeelementen

Programmspeicher

Ein Mikrocontroller muss sein übersetztes Anwendungsprogramm in einem nichtflüchtigen On-Chip-Programmspeicher haben. Größe der Speicher: wenige Bytes bis zu mehreren Mbyte.

Flash-EEPROM Häufigster Fall! Der Flash-Speicher wird entweder im Entwicklungssystem (über USB-JTAG) geflasht oder über eine serielle Verbindung

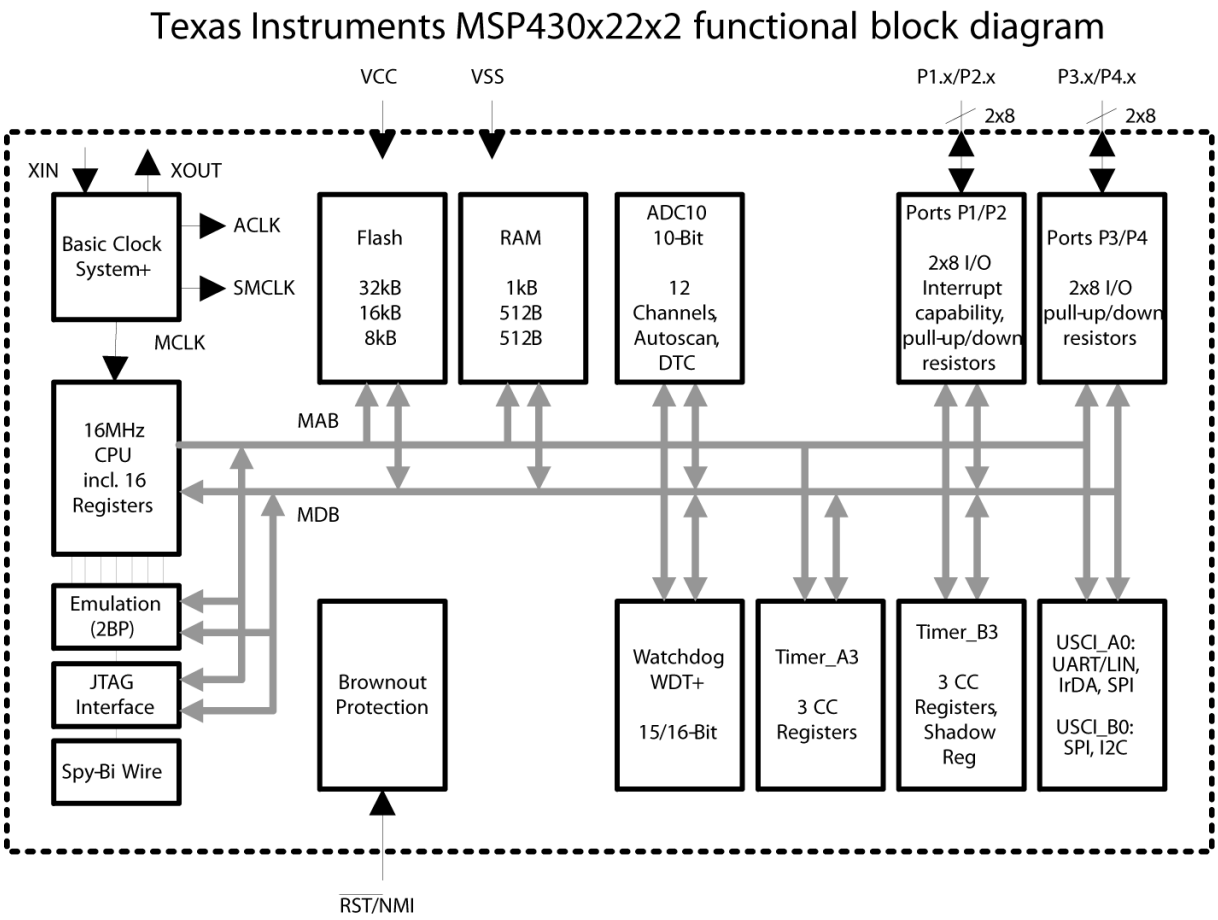


Figure 2.3: Blockdiagramm des MSP2272 von Texas Instruments. (Abb. mit freundlicher Genehmigung von Texas Instruments).

durch einen Bootloader; auch möglich wenn im System eingebaut. *Feldprogrammierbarer Mikrocontroller* oder auch *In System Programming-Mikrocontroller* (ISP-MC).

EPROM und EEPROM Der Mikrocontroller kann beim Kunden programmiert werden, das Programm bleibt änderbar; geeignet für Entwicklung und Test der Programme.

OTP-ROM Der Mikrocontroller kann einmal beim Kunden programmiert werden; geeignet für Kleinserien.

Masken-ROM Programmcode wird bei der Herstellung des Mikrocontroller eingearbeitet (Maske), und ist nicht mehr änderbar; geeignet für Großserien.

Nicht-flüchtiges RAM (nonvolatile RAM, NV-RAM) Der Inhalt der Speicherzellen wird vor dem Ausschalten in EEPROM-Zellen übertragen. (selten)

Mikrocontroller, die ausschließlich mit einem externen Programmspeicher arbeiten sollen, werden als ROM-lose Controller auch ganz ohne internes ROM gefertigt.

Datenspeicher

Register Gruppen von Flipflops mit gemeinsamer Steuerung innerhalb CPU Zugriff ohne Bustransfer, daher sehr schnell. Compiler versuchen, Daten möglichst hier abzulegen. Breite der Register: 4/8/16/32 Bit

Allgemeiner Datenbereich On-Chip-Datenspeicher, meist SRAM, (da kein Refresh nötig), Zwischenspeicherung von Programmdaten (Variablen) die nicht mehr in die Register passen

Stack Teil des allgemeinen Datenbereichs, kurzzeitige Zwischenspeicherung von Daten, Zugriff mit PUSH und POP.

Special Function Register Konfiguration und Ansteuerung der Peripheriebereiche des Mikrocontrollers,

Variante: Registerbänke Zwischen mehreren Registerbänken kann umgeschaltet werden, schneller Kontextwechsel

Variante: Bitadressierbarer Bereich Ein Bereich mit direktem Bitzugriff

Die Größe der On-Chip-Datenspeicher: Wenige Bytes bis zu einigen KByte, meist deutlich kleiner als der Programmspeicher. Für Daten, die ohne Spannungsversorgung erhalten bleiben sollen, besitzen manche Controller einen Flash-, EEPROM- oder NVRAM-Bereich. Datenbereiche:

Evaluation-Boards enthalten auf einer Platine außer dem Mikrocontroller auch etwas Peripherie. Mit einem Evaluation-Board kann schnell und preiswert ein bestimmer Controller getestet werden, ohne dass vorher eine Platine gefertigt werden muss.

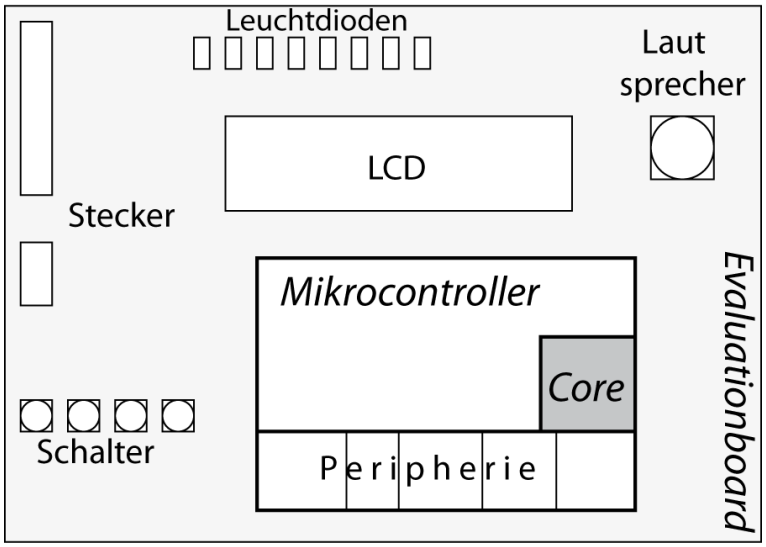


Figure 2.4: Ein Evaluation-Board enthält den Mikrocontroller und etwas Peripherie zum Austesten des Controllers.

Übung: 2.1 Verwendung verschiedener Speicher

Warum besitzen Mikrocontroller verschiedene Arten von Speicher (ROM, RAM, Register...) und wozu verwendet man sie?

Übung: 2.2 Speicherauswahl bei Geräteentwicklung

Auf der Basis eines Mikrocontrollers soll ein Kleingerät entwickelt werden. Es ist geplant, später pro Jahr 100000 Stück zu fertigen. Welchen Typ Programmspeicher sehen Sie für die verschiedenen Entwicklungs- und Fertigungsphasen vor?

2.2 Kurzeinführung: Der MSP430 von Texas Instruments

Überblick

Wir benutzen in diesem Versuch ein Board, auf dem ein MSP430F2272 von Texas Instruments als zentraler Baustein arbeitet. Diese Mikrocontroller-Familie hat eine moderne Architektur, bietet genug Literaturquellen und ist mit seinem energiesparenden Design zukunftssicher. In Stichworten:

- 16-Bit RISC CPU
- 16 Allzweck-Register zu 16 Bit
- Taktfrequenzen von 25 kHz bis zu 16 MHz
- kompakter Kern, energiesparendes Design (fünf Low-Power-Modi)
- zwischen 1kB und 256 kB Flash, auf Chip programmierbar
- bis zu 16 kB RAM
- Zahlreiche Peripheriegruppen verfügbar, u.a. 10, 12 oder 16-Bit-Analog/Digital-Wandler, Digital/Analog-Wandler, LCD-Treiber, Überwachung der Versorgungsspannung, flexibles Interrupt-System, Operationsverstärker, Zähler und Zeitgeber, Watchdog-Timer, Brown-Out-Detektor, serielle Schnittstelle, I2C, SPI, IrDA, Hardwaremultiplier, DMA-Controller

Die Register

Der MSP430 besitzt 16 Register, die ersten vier davon sind für spezielle Aufgaben reserviert:

- R0 = Program Counter, hält die Adresse des nächsten auszuführenden Befehls
- R1 = Stack Pointer, verweist auf den Stack.
- R2 = Status Register, enthält die Zustandsinformationen des Prozessors
- R3 = Konstanten-Generator
- R4 – R15 Allzweck-Register, frei benutzbar für Daten.

Name MSP=Mixed Signal Processor

3 Software-Entwicklung für Mikroprozessoren (Teil I)

3.1 Umgang mit der Dokumentation

Mikrocontroller bilden Familien: Gleicher Kern, verschiedene Peripherie- und Speicherausstattung.

Wir benutzen den Mikrocontroller: TI MSP430F2272
Bei uns ist also die Familie: Texas Instruments MSP430 und der Typ: F2272

Es gibt verschiedene Dokumentationen (alles .pdf):

Der MSP430Fx2xx Family User's Guide enthält alle allgemeinen Informationen über das Funktionieren der Baugruppen.

Benutzen Sie den MSP430Fx2xx Family User's Guide , wenn die gesuchte Information für alle Mitglieder der Familie gültig ist. Beispiel: Funktionsweise von Timer A.

Das MSP430x22x2/22x4 Datasheet enthält alle speziellen Informationen, die nur für diese Typen gültig sind. (MSP430x2232, MSP430x2252, MSP430x2272, MSP430x2234, MSP430x2254, MSP430x2274)

Benutzen Sie das MSP430x22x2/22x4 Datasheet, wenn die gesuchte Information nur für den betreffenden Device (Chip) gilt. Beispiel: Funktion der Anschlussleitungen (Pins)

3.2 Hochsprache, Assemblersprache und Maschinencode

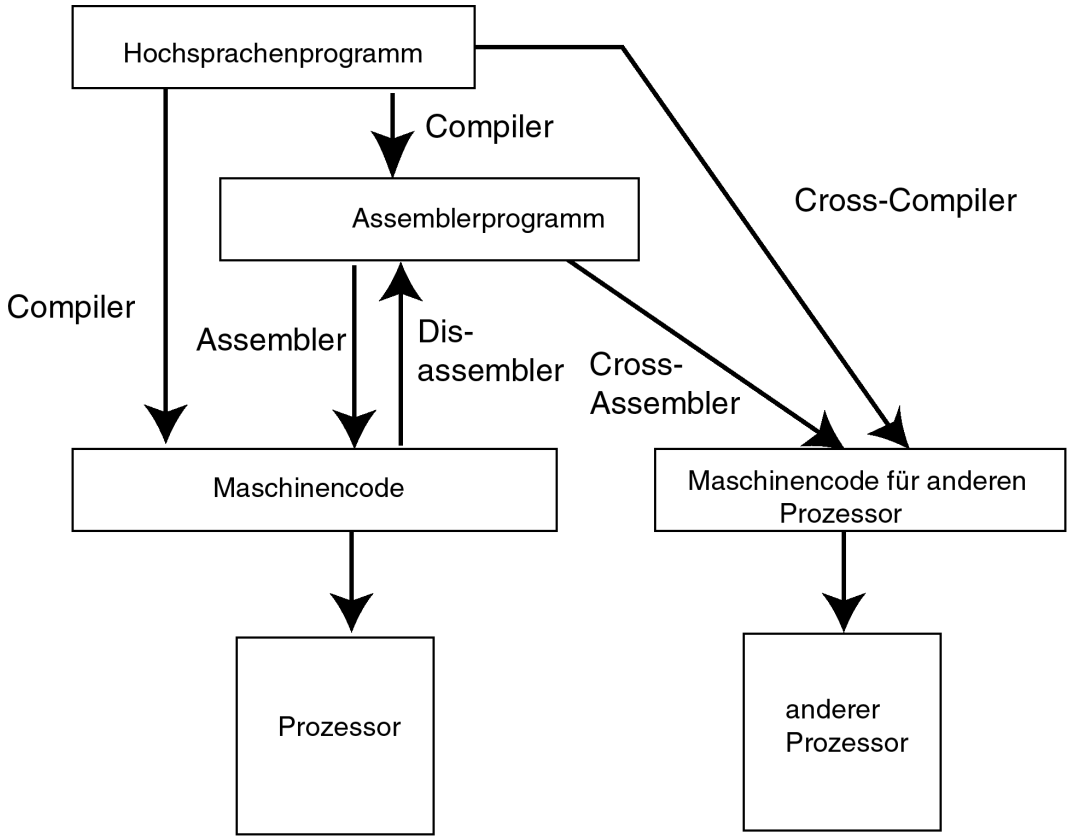


Figure 3.1: Der Maschinencode für einen Mikroprozessor/Mikrocontroller kann durch verschiedene Übersetzungswerkzeuge erzeugt werden.

Einführendes Beispiel (MSP430) Im C-Quellcode steht der C-Befehl

```
Zaehler = 7;
```

Wenn wir annehmen, dass die Variable Zaehler vorübergehend in Register R12 gespeichert ist, wird das vom Compiler in den folgenden gut lesbaren Assemblercode übersetzt:

```
mov.w #0x0007, R12
```
("'Move word (16 Bit) constant 7 to Register Nr.12'")

Daraus wird der Maschinencode:

```
403Ch 0007h
```

Dabei ist das erste 16-Bit Wort der OpCode 403Ch, 40 steht für Move und C ist die Nummer des Registers (12). "'0007"' ist der (konstante) 16-Bit-Operand, der dem Opcode folgt.

Viele Entwicklungsumgebungen haben einen Debugger, der auf Wunsch auch den Assembler- und Maschinencode zeigt.

Programmieren in Maschinencode

Der Prozessor selbst verarbeitet nur Maschinencode, aber das Arbeiten mit Maschinencode hat einige schwere Nachteile:

- Die Programme sind sehr unflexibel und schwer änderbar.
- Die Programme sind sehr schlecht lesbar, man kann die Maschinenbefehle nicht erkennen und keine Namen für Variablen und Sprungmarken vergeben.
- Es können keine Kommentare eingefügt werden.

Programmieren in Assemblersprache

Die natürliche Programmierpsrache für einen Mikroprozessor ist Assemblersprache. Was ist nun Assemblersprache?

- Die Assemblersprache ist eine Programmiersprache, die das direkte Programmieren in Maschinensprache ermöglicht
- Die Maschinenbefehle werden eingängige Abkürzungen dargestellt, sogenannte *Mnemonics*, z.B. "'ADD"' für Addiere, MOV für "'MOVE"', CLC für "'Clear Carryflag"' usw.
- Assemblersprache ist eine 1:1-Abbildung des Maschinenbefehlssatzes.
- Mit einem Disassembler kann umgekehrt Maschinencode umgewandelt und gut lesbar in Assemblersprache dargestellt werden.

Vorteile der Assemblerprogrammierung

- Programme in Assemblersprache ermöglichen die perfekte Kontrolle über die Prozessorhardware
- Programme in Assemblersprache ermöglichen optimale Performance.

Nachteile der Assemblerprogrammierung

- Programme sind schwerer zu verstehen als in Hochsprache
- Programme sind aufwändiger zu erstellen und zu ändern
- Fehler sind schwerer zu finden
- Jeder Prozessor hat seine eigene Assemblersprache
- Bei sehr mächtigen Prozessoren: Optimierungsmöglichkeiten des Compilers fehlen

Programmieren in Hochsprache

Deshalb werden auch Mikroprozessoren heute überwiegend in Hochsprachen programmiert. Aus der Hochsprache wird dann direkt oder auf dem Umweg über Assemblercode der Maschinencode erzeugt. Übliche Hochsprachen sind:

- C (ganz überwiegend)
- C++ (langsam zunehmend, aber noch nicht verbreitet)

- Basic
- Java
- Pascal

Viele Entwicklungsumgebungen ermöglichen die Verwendung von Assembler in C-Programmen, es gibt zwei Wege:

- Man mischt Assemblermodule (Quelldateien) und Hochsprachenmodule. Dabei werden die Module getrennt übersetzt und danach der Maschinencode zusammen gelinkt.
- Man streut in ein Hochsprachenprogramm Assemblerbefehle ein, der so genannte *Inline-Assembler*.

Dies ist nützlich oder nötig wenn

- ein Codeabschnitt besondere Hardwarenähe hat (z.B. bei Speicherverwaltung initialisieren)
- die Performance optimiert werden soll (Innerer Teil der Schleifen)

3.3 Wertebereich von Variablen

Der Wertebereich von Programmvariablen hängt davon ab, wie viele Bit für die Speicherung der Variablen zur Verfügung stehen! Zur Erinnerung: Wenn für eine Variable N Bit zur Verfügung stehen, ergibt sich ein Wertebereich von

- Vorzeichenlos: $0 \ldots + 2^N - 1$
- Mit Vorzeichen: $-2^{N-1} \ldots + 2^{N-1} - 1$

Beispiele:

Speicherplatz	Wertebereich vorzeichenlos	Wertebereich mit Vorzeichen
8 Bit	0 ... 255	-128 ... +127
16 Bit	0 ... 65535	-32768 ... +32767
32 Bit	0 ... 4294967295	-2147483648 ... +2147483647

Eine Überschreitung dieser Grenzen führt zu dramatischen Programmfehlern! Man muss also bereits bei der Programmierung darauf achten, dass genügend Bit zur

Verfügung stehen. Also: Richtigen Datentyp wählen!

Übung: 3.1 Übung zu den Wertebereichen

a) Ein Schleifenzähler soll sich im Bereich von -3 bis +1000 bewegen. Welchen Datentyp wählen Sie?

b) Mit drei Variablen A,B,C soll C=A+B, und später C=A*B berechnet werden. A und B können Werte zwischen 0 und 1023 annehmen. Wählen Sie den richtigen Datentyp für C.

MEHR INFORMATIONEN

→ MSP430 Family's User Guide, Abschnitt 16-Bit RISC CPU / Instruction Set

→ Buch: Mikrocontrollertechnik

→ Buch: Das große MSP430-Praxisbuch

4 Digitale Ein- und Ausgabe

4.1 Allgemeine Funktionsweise

Die digitale Ein- und Ausgabe ist die klassische Methode, um Signale bzw. Daten zwischen digitalen Systemen auszutauschen. Auf den Leitungen gibt es nur HIGH-/LOW. Andere Bezeichnungen:

- Eingabe/Ausgabe, E/A
- Input/Output, I/O, IO, Digital-IO
- I/O-Ports (port = Hafen, Anschluss) oder einfach Ports

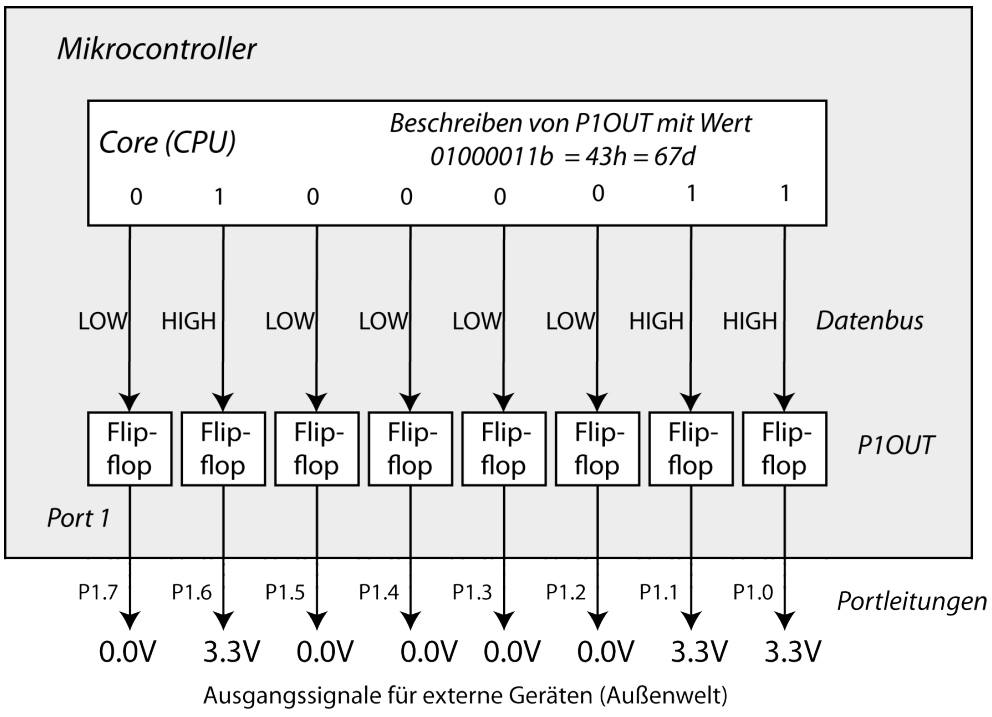


Figure 4.1: Ausgabevorgang (Output).

Ausgabe:

- Ein Bit im Programm wird in einen Pegel auf der Portleitung umgesetzt
- Aus 0 wird LOW
- Aus 1 wird HIGH

Durch Ausgabe kann also abhängig von den Variablen eines Programms eine Ausgangsleitung geschaltet werden, an der wiederum ein elektrisches Gerät hängen

kann. So kann ein Mikroprozessor/Mikrocontroller ein externes Gerät steuern, wie z. B. eine Leuchtdiode oder einen Motor.

Eingabe:

- Ein Pegel auf einer Portleitung wird in ein Bit des Programms umgesetzt
- Aus LOW wird 0
- Aus HIGH wird 1

Damit kann der Zustand eines externen Gerätes erfasst und in eine Programmvariable umgesetzt werden. Auf diese Art kann das Programm in dem Mikrocontroller Information von einem externen Gerät erhalten, z.B. einem Schalter oder einem anderen Controller.

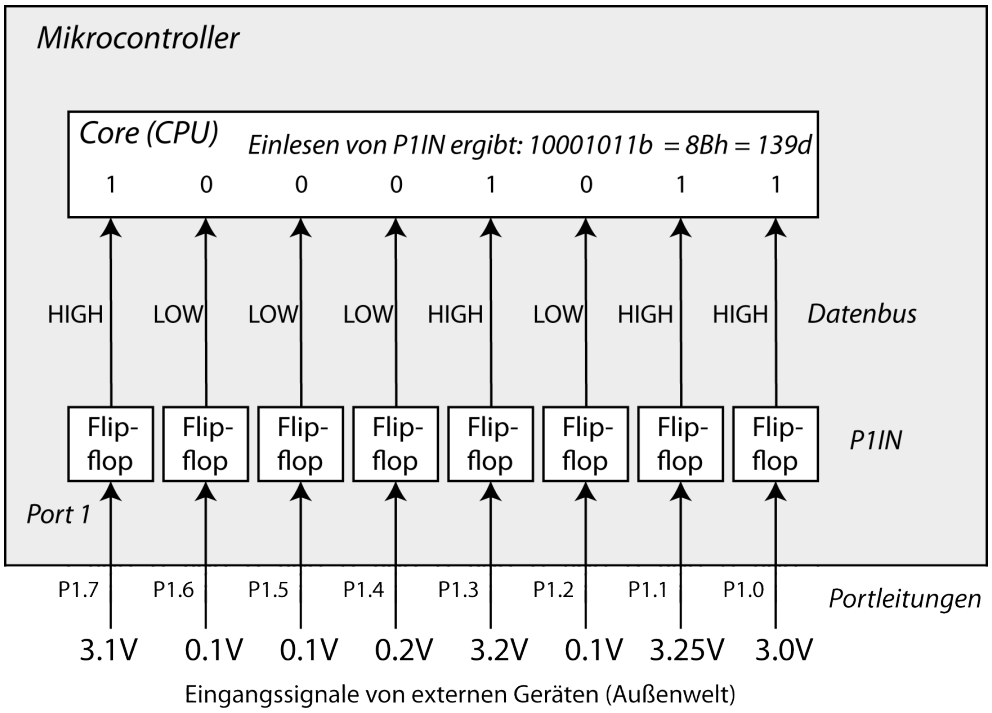


Figure 4.2: Eingabevorgang (Input).

Schaltung für Ein- und Ausgabe

Eine etwas flexiblere Schaltung, die wahlweise für Eingabe oder Ausgabe benutzt wird, ist in Abb. 4.3 gezeigt. In das Flipflop1 wird zunächst die Richtung eingetragen: 1=Eingabe, 0=Ausgabe. Ist die Richtung "'Eingabe"' gewählt, kann durch das Signal "'Eingabewert übernehmen"' das an der Ein-/Ausgabeleitung anliegende Signal im Flipflop3 eingespeichert werden. Durch das Steuersignal "'Eingabe"' wird es auf den Datenbus eingekoppelt. Ist dagegen die Richtung "'Ausgabe"' gewählt, so wird der in Flipflop2 gespeicherte Wert in ein TTL-Signal gewandelt und dauerhaft auf die Ein-/Ausgabeleitung gegeben. Mit dem Steuersignal "'Ausgabewert schreiben"' kann in Flipflop2 ein neuer Wert eingetragen werden.

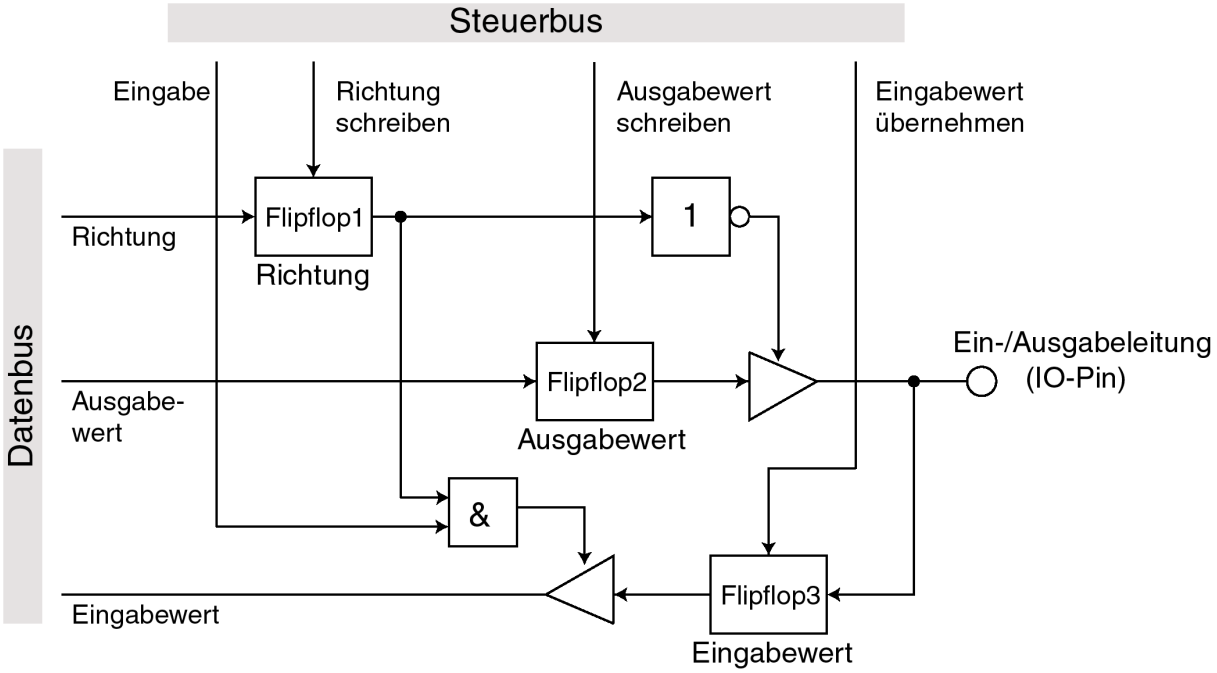


Figure 4.3: Eine flexible Schaltung, die für Ein- und Ausgabe geeignet ist.

4.2 Digitale Ein-/Ausgabe bei Mikrocontrollern

- Für Mikrocontroller besonders wichtig, Austausch *digitaler Signale* mit dem umgebenden System. Beispiele: Ansteuern LED (Ausgabe), Einlesen Schalter (Eingabe)
- Alle Mikrocontroller haben mehrere IO-Ports
- meist in Gruppen zu 8 Bit organisiert.
- Port hat meistens Datenregister und Richtungsregister (Eingabe oder Ausgabe?)
- oft rücklesbar, im Ausgabebetrieb kann Wert vom Controllerkern wieder eingelesen werden.
- Schaltungstechnik: Ausgangsstufen sind Open-Collector-, Open-Drain- oder Gegentakt-Endstufen.

Fallbeispiel: Digitale IO-Leitungen beim MSP430

Übersicht

Beispiel zu Bild 4.4, Pin 8 hat drei Funktionen:

1. Leitung 0 vom Allzweck-Digital-IO-Port 2 (P2.0)
2. Ausgabeleitung für Auxiliary Clock (ACLK)
3. Kanal 0 des Analog/Digital-Wandlers (A0)

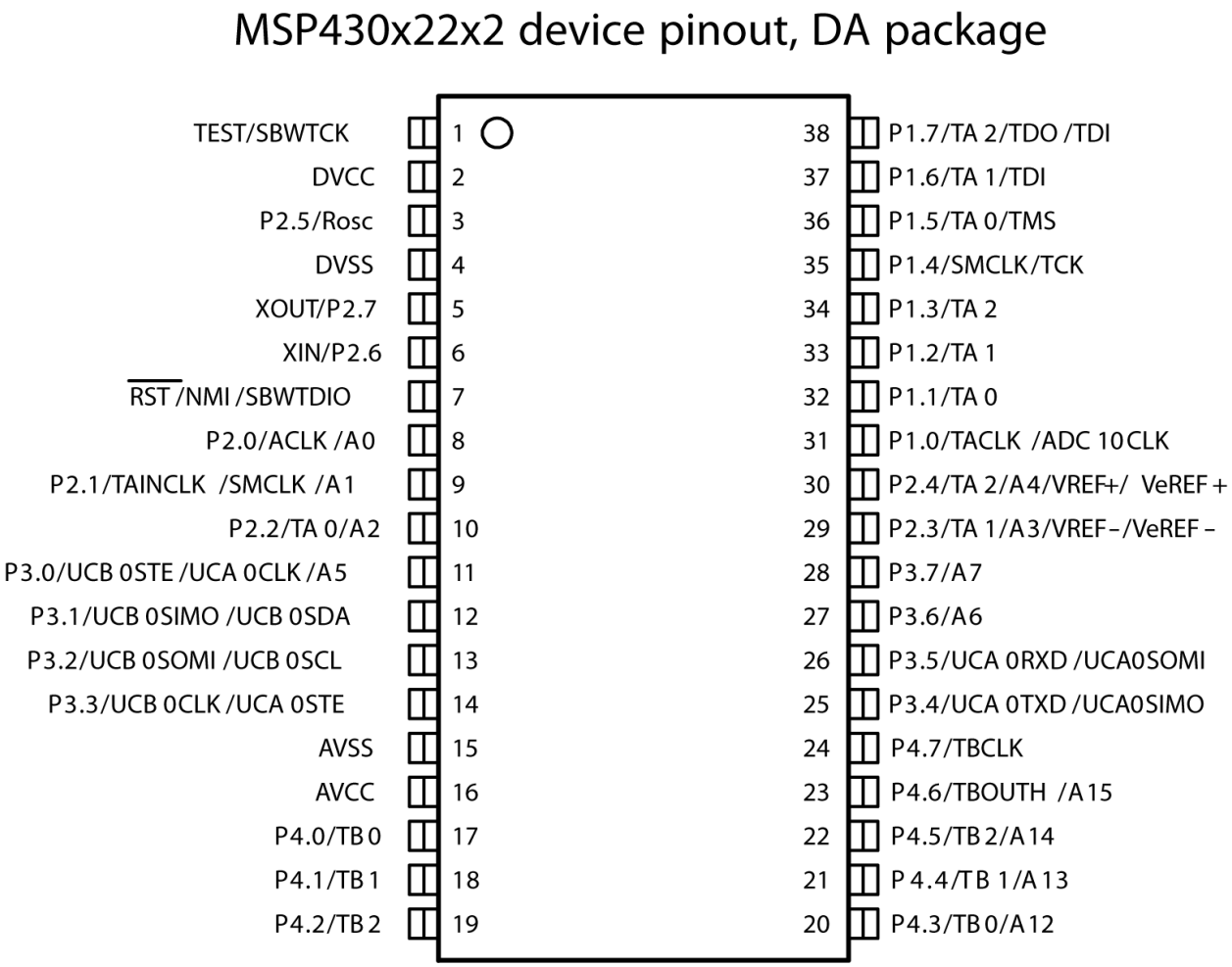


Figure 4.4: Pinout (Belegung der Anschlussstifte) des MSP2272 von Texas Instruments. Alle Anschlussstifte (Pins) sind mit mehreren Funktionen belegt. (Abb. mit freundlicher Genehmigung von Texas Instruments).

Auswahl der Funktion: Über die Select-Register (PxSEL und evtl. PxSEL2) wird die einfache IO-Funktion (Voreinstellung) oder eine alternative Funktion (hinter dem Schrägstrich) gewählt. Näheres: Datasheet.

- Die digitalen Ein-/Ausgänge (I/O-Ports, oder einfach Ports) sind mit P1, P2 usw. bezeichnet
- Jeder Port hat 8 Leitungen
- Jede Leitung kann unabhängig als Ein- oder Ausgang eingestellt werden
- Die Ports P1 und P2 sind interruptfähig
- Zu jedem IO-Port gehören mehrere Konfigurations-Register
- Die Konfigurations-Register der IO-Ports haben 8 Bit
- Die 8 Bit sind den 8 Leitungen zugeordnet, also Bit 0 gehört jeweils zu Leitung 0, Bit 1 zu Leitung 1

Betrachten wir beispielhaft die Register für Port P1:

P1IN, das Eingangspufferregister Wenn zugehörige Eingangsleitung LOW ist: Bit=0; wenn Eingangsleitung HIGH ist: Bit=1.

P1OUT, das Ausgangspufferregister Bit=0: zugehörige Ausgangsleitung wird auf LOW geschaltet; Bit=1: Leitung wird HIGH.

P1DIR, das Richtungsregister Bit=0: zugehörige Leitung ist Eingang; Bit=1: zugehörige Leitung ist Ausgangsleitung

P1REN, das Resistor Enable Flag Register Aktiviert die eingebauten Pull-Up- oder Pull-Down-Widerstände. Diese ziehen offene Eingänge auf definierte Potentiale.

P1SEL, das Port Select Register Bit=1 hier aktiviert für diese Leitung die erste alternative Leitungsfunktion.

P1SEL2, das Port Select 2 Register Bit=1 (und im Register P1SEL) aktiviert für diese Leitung die zweite alternative Leitungsfunktion.

P1IFG, das Interrupt Flag Register Bit=1 zeigt an, ob über diese Leitung ein Interrupt ausgelöst wurde.

P1IE, das Interrupt Enable Register Bit=1: Interruptauslösung für die zugehörige Leitung ist aktiviert

P1IES, das Interrupt Edge Select Register Auswahl, ob der Interruptauslösung bei ansteigender Flanke (Wechsel von LOW auf HIGH) oder bei fallender Flanke

Table 8–1. Digital I/O Registers

Port	Register	Short Form	Address	Register Type	Initial State
P1	Input	P1IN	020h	Read only	–
	Output	P1OUT	021h	Read/write	Unchanged
	Direction	P1DIR	022h	Read/write	Reset with PUC
	Interrupt Flag	P1IFG	023h	Read/write	Reset with PUC
	Interrupt Edge Select	P1IES	024h	Read/write	Unchanged
	Interrupt Enable	P1IE	025h	Read/write	Reset with PUC
	Port Select	P1SEL	026h	Read/write	Reset with PUC
	Port Select 2	P1SEL2	041h	Read/write	Reset with PUC
	Resistor Enable	P1REN	027h	Read/write	Reset with PUC

Figure 4.5: Die IO-Ports werden über eine Gruppe von Registern gesteuert, hier sind die Register für Port 1 gezeigt. (Family User's Guide zum MSP430 mit freundlicher Genehmigung von Texas Instruments)

Wenn wir einen Blick in die Dokumentation von Texas Instruments werfen, erhalten wir über diese Register weitere Information.[48] Einen Auszug haben wir in Abb. 4.5 dargestellt. In der vierten Spalte finden wir die Hardware-Adresse des entsprechenden Registers. In Spalte 5 ist angegeben, ob das Register nur gelesen (read only) oder auch beschrieben werden kann (read/write). (Letzte Spalte: Zustand direkt nach Einschalten, PUC = "'Power Up Clear'")

Entscheidung beim Betrieb einer IO-Leitung:

- Einfaches IO beabsichtigt? (Voreinstellung)

- Oder alternative Peripheriefunktion? (siehe Abb.4.4)
- Wenn einfaches IO: Soll Leitung Ein- oder Ausgang ?
- Interruptauslösung gewünscht?
- Wenn Interruptauslösung, mit welcher Flanke? (LOW→HIGH–Transition) oder (HIGH→LOW–Transition ?)

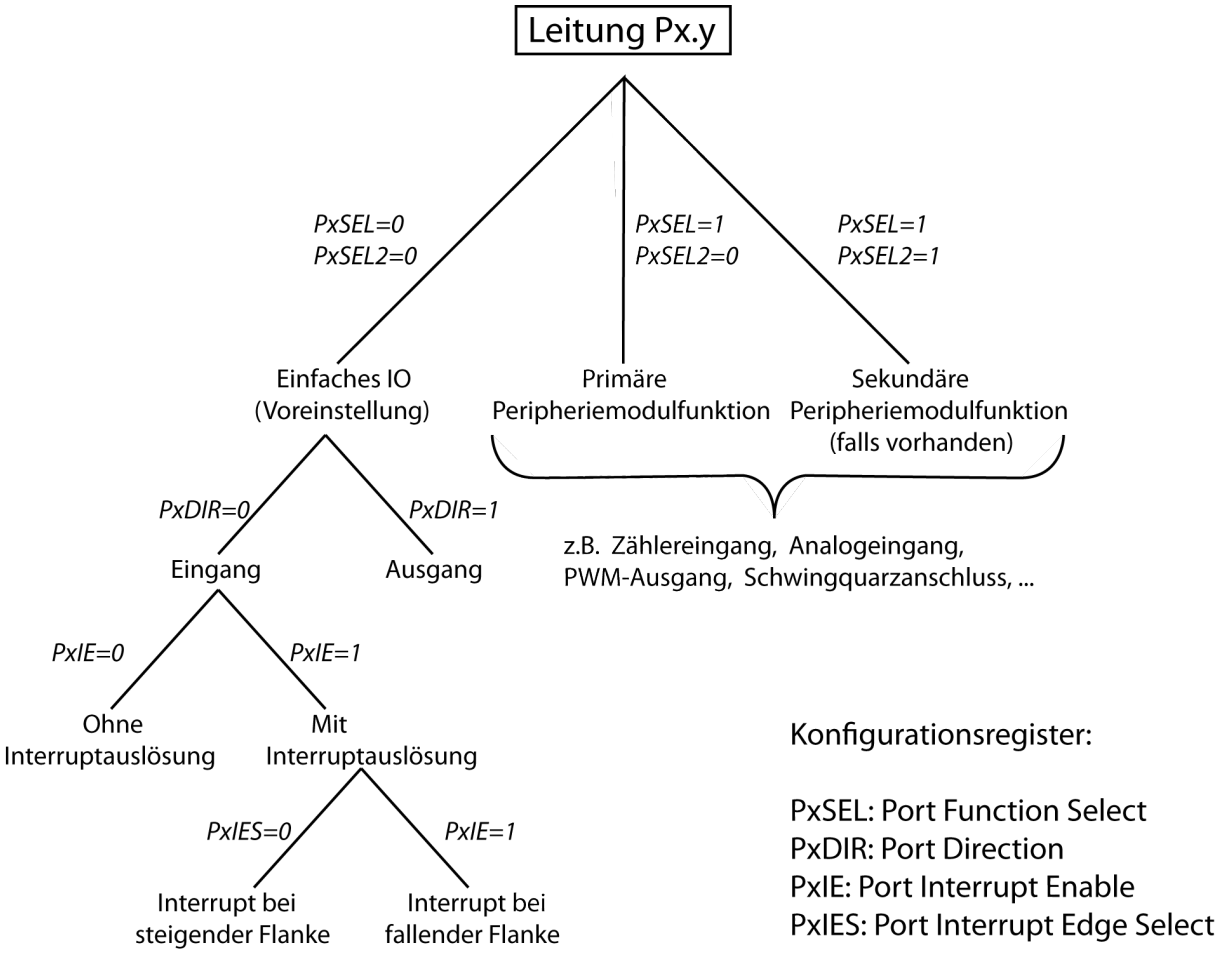


Figure 4.6: Die IO-Leitungen (Pins) können für verschiedene Funktionen benutzt werden. das muss vor der Benutzung über die Konfigurationsregister eingestellt werden.

Welche Pegel als HIGH und welche als LOW erkannt werden, hängt von der Betriebsspannung ab (s. Datasheet) Dort sind auch alle Funktionen der Pins beschrieben. Ein MSP430 hat bis zu 8 IO-Ports. Port 2 funktioniert genau wie Port 1, die Steuerregister heißen hier P2IN, P2OUT usw. Port 1 und Port 2 sind interruptfähig, ab Port 3 ist es daher einfacher: Register PXIFG, PXIE und PXIES gibt es hier nicht.

Einfaches IO am MSP430 in Stichworten:

- Initialisierung: PxDIR (x ist die Portnummer) und falls gewünscht PxREN beschreiben.
- Jetzt kann man über PxIN und PxOUT den Port benutzen

- Falls Interrupt-Betrieb gewünscht: Den Interrupt aktivieren, auslösende Flanke festlegen, Interrupt-Handler schreiben.

Beispiel Ein Programm, das eine Leuchtdiode blinken lässt, die an der Leitung 0 von Port 1 angeschlossen ist, also P1.0. Mit dem Watchdog Timer wollen wir uns hier noch nicht befassen und schalten ihn ab. Der Endwert in den beiden Zählschleifen ist einfach durch Ausprobieren gefunden worden.

```
/* ************************************************************
  Beispielprogramm blink1.c
  Lässt auf dem Board eine LED endlos blinken

  Kommentar zur Schaltung auf Board:
  Leuchtdioden an P1.0 - P1.7 leuchten wenn Ausgang=L (0)

*/

#include <msp430x22x2.h>  // Header-Datei mit den
                          // Hardwaredefinitionen für genau diesen MC

int main(void) {

  WDTCTL = WDTPW + WDTHOLD;          // watchdog timer anhalten

  // Hardware-Konfiguration
  P1DIR = 0x01;     // Control Register P1DIR beschreiben:
                    // Leitung 0 wird Ausgang, die anderen Eingänge

  while (1)  {                       // Endlosschleife (Round-Robin)
    P1OUT=0x01;                      // LED an Port1.0 ausschalten
    for (i = 50000; i > 0; i--);     // Warteschleife
    P1OUT=0x00;                      // LED an Port1.0 einschalten
    for (i = 50000; i > 0; i--);     // Warteschleife
  }

 return 0;    // Statement wird nicht erreicht
}
```

Auslesen von Schaltern und Tastern

Schalter und Taster sind das MMI (Man-Machine-Interface) für die Eingabe, damit lassen sich Mikrocontroller von Bedienern steuern. Sie werden meistens an Digitaleingänge angeschlossen. Beim Auslesen von Tastern (Drück-Schalter mit nur einer stabilen Stellung) muss man zunächst heraus finden, ob sie in der Ruhestellung HIGH oder LOW liefern.

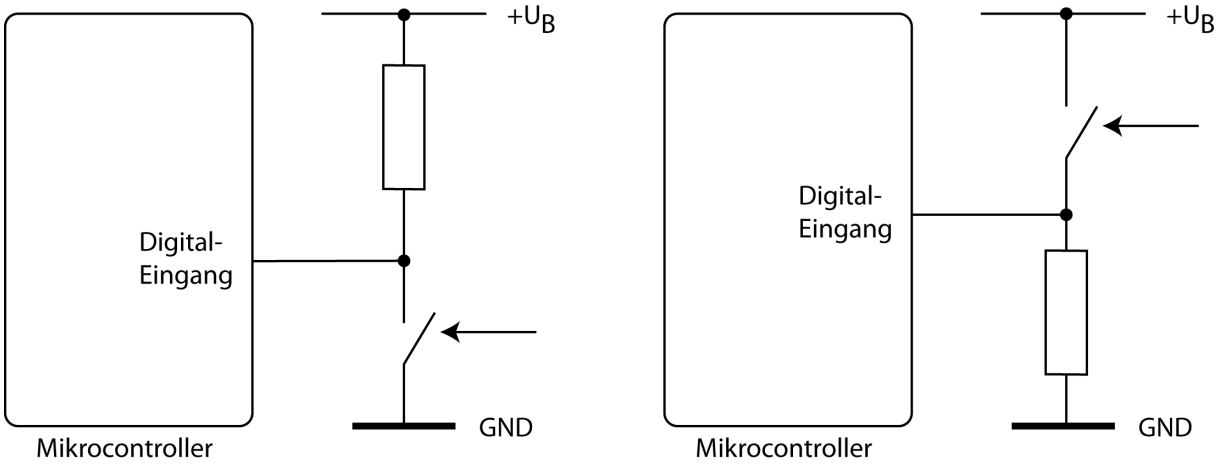


Figure 4.7: Zwei Möglichkeiten einen Schalter an einen Digitaleingang anzuschließen. Schaltung links: Eingang ist HIGH, wenn Taster nicht gedrückt und LOW wenn gedrückt. Schaltung rechts: Umgekehrt.

Prellen: Wenn der Taster gedrückt wird, schließt er nicht sauber in einem Schritt, sondern schließt mehrfach kurz, bis er endgültig Verbindung hat.

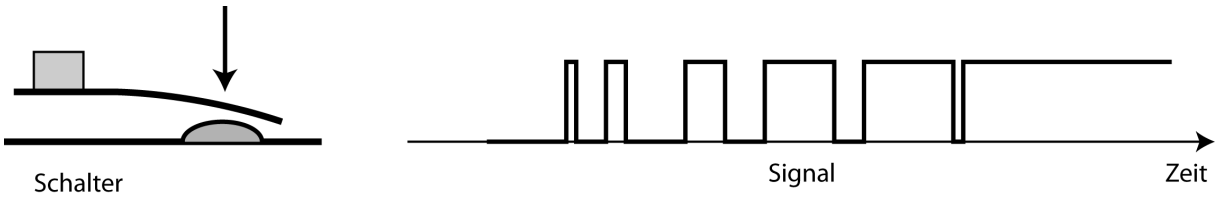


Figure 4.8: Praktisch alle Schalter prellen, sie kontaktieren also mehrfach, bis der sichere Kontakt (Schluss) besteht.

Taster und Schalter kann man softwaremäßig entprellen. Ein anderes Problem ist die Mehrfacherkennung: Bei der Bedienung eines Tasters wird der Taster für längere Zeit gedrückt sein. In dieser Zeit kann die Software mehrfach den gedrückten Schalter erkennen. Das ist aber erst nach einiger Zeit als Auto-Repeat gewünscht. Steuerung: Software.

MEHR INFORMATIONEN

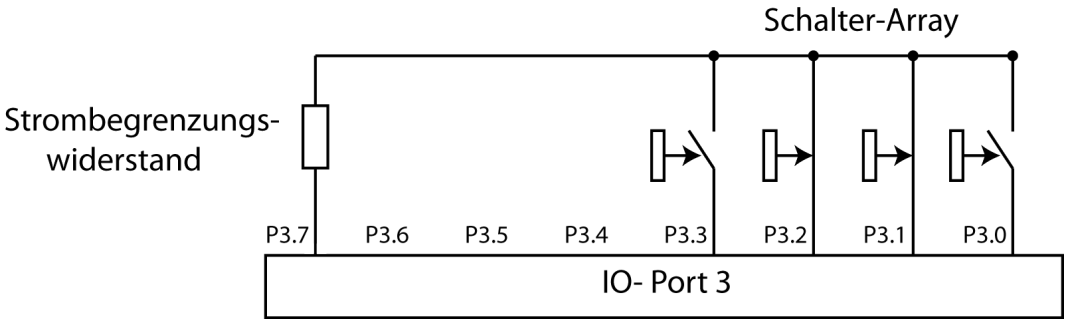
→ MSP430 Family's User Guide, Abschnitt Digital I/O

→ Buch: Mikrocontrollertechnik, Abschnitt "'Die parallelen Schnittstellen"'

→ Buch: Das große MSP430-Praxisbuch, Abschnitt "'Die digitalen Ein-Ausgabeports"'

Übung: 4.1 Schalter einlesen

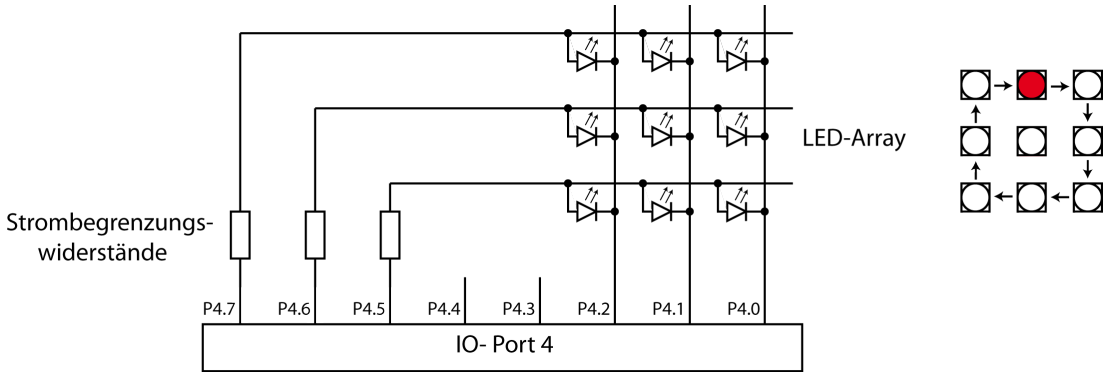
Betrachten Sie die unten dargestellte IO-Schaltung.



a) Wie muss Port 3 konfiguriert werden, um die Schalter auszulesen?
b) Welches Bitmuster (= Zahlenwert) wird man im Datenregister von Port 3 lesen, wenn zwei der Schalter gedrückt sind, wie im Bild dargestellt.
c) Schreiben Sie den vollständigen C-Code auf, um den Schalter auszulesen und das Ergebnis auf die Variable "'Switch1"' zu Übertragen!

Übung: 4.2 LED-Array ansteuern

Betrachten Sie die unten dargestellte IO-Schaltung.



a) Wie muss Port 4 konfiguriert werden, um das LED-Array anzusteuern?
b) Welche Bitmuster (=Zahlenwerte) müssen auf den Port 4 geschrieben werden, um ein im Uhrzeigersinn umlaufendes Licht zu erzeugen?
c) Schreiben Sie den vollständigen C-Code auf, um das Licht einmal umlaufen zu lassen!

4.3 Ein- und Ausgabe in Desktop-Rechnern

- Der zentrale Bereich eines Rechnersystems ist das Prozessor-Hauptspeicher-Cache-System.
- Externe Geräte wie Tastatur, Maus, Laufwerke, Echtzeituhr, Grafikkarte, USB-Host usw. müssen anders angesprochen werden.
- Das Gleiche gilt für die Systembausteine auf der Hauptplatine, wie Interrupt-

controller, DMA-Controller, Zeitgeberbaustein u.a.m.

- Alle diese Komponenten werden über *Eingabe* und *Ausgabe* (Input und Output) angesprochen.
- Die I/O-Bausteine sind oft in andere Bausteine integriert, z.B. in Controller und Schnittstellen.

Diese Subsysteme arbeiten unabhängig vom Hauptprozessor: eigene Signalpegel und eigene Protokolle. Sie müssen regelmäßig Daten mit dem Prozessor-Hauptspeicher-System austauschen. Die IO-Schaltungen sind hier in die Subsysteme bzw. deren Controller (Steuerungen) integriert. Dazu gibt es typischerweise drei Gruppen von Registern:

Zustandsregister geben Informationen über den Zustand des Geräts, lesender Zugriff,

Steuerregister steuern die Funktionsweise der Baugruppe, schreibender Zugriff

Datenregister halten vorübergehend Daten, die weiter gereicht werden, lesender oder schreibender Zugriff je nach Richtung

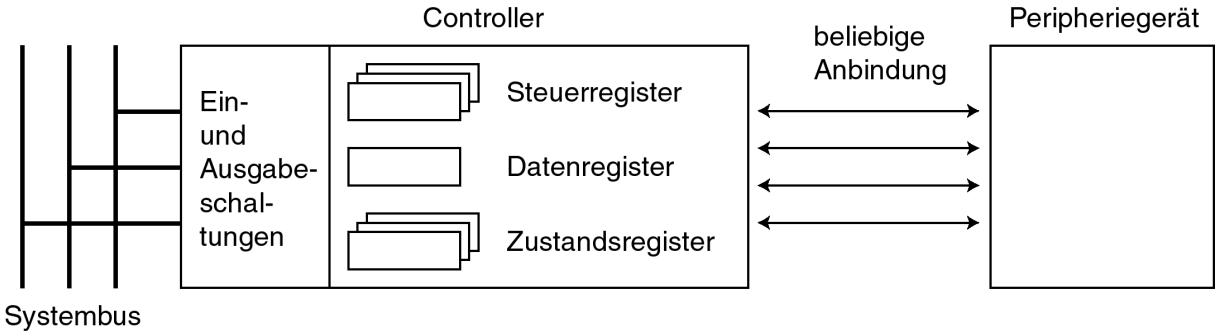


Figure 4.9: Ein Peripheriegerät bildet mit seinem Controller ein Subsystem. Die Anbindung an den Systembus erfolgt durch Register, auf die über Ein-/Ausgabeschaltungen zugegriffen wird.

Beispiel:
Der Prozessor möchte ein Datenwort über die serielle RS232-Schnittstelle senden.

- Prozessor schreibt Betriebsparameter der Übertragung auf Steuerregister der Schnittstelle (Ausgabe).
- Statusregister der Schnittstelle auslesen um festzustellen, ob die Serialisierungseinheit frei oder belegt ist (Eingabe)
- Datenwort in das Datenregister der Schnittstelle schreiben (Ausgabe)
- Schnittstelle sendet

Durch Folgen von Ein- und Ausgabevorgängen können auch komplexe Geräte und Schnittstellen angesteuert werden. Die Software, die das ausführt, wird im Betriebssystem als so genannter *Gerätetreiber* (device driver) geführt. Der Gerätetreiber sorgt auch für die Synchronisation, Prozessor und Gerät haben ja völlig unterschiedliches Timing.

5 Software-Entwicklung für Mikroprozessoren (Teil II)

5.1 Entwicklungsumgebung

Programmentwicklung mit Zielsystem PC:

- Programmentwicklung findet auf einem PC statt
- Zielsystem hat Bildschirm, Tastatur, Maus, Laufwerke
- erzeugter Maschinencode kann direkt auf dem Entwicklungsrechner ausgeführt werden

Programm-Entwicklung für anderen Mikroprozessor oder -controller:

- Der Mikrocontroller kann nicht als Entwicklungssystem benutzt werden, weil er in der Regel weder Bildschirm noch Tastatur hat.
- Ein Mikrocontroller wird minimal ausgewählt, auf dem System ist kein Platz für Entwicklungswerkzeuge.
- Erzeugung des Maschinencodes auf dem Entwicklungssystem, dort nicht lauffähig (cross-platform-Entwicklung)
- Weniger Möglichkeiten zur Fehlersuche

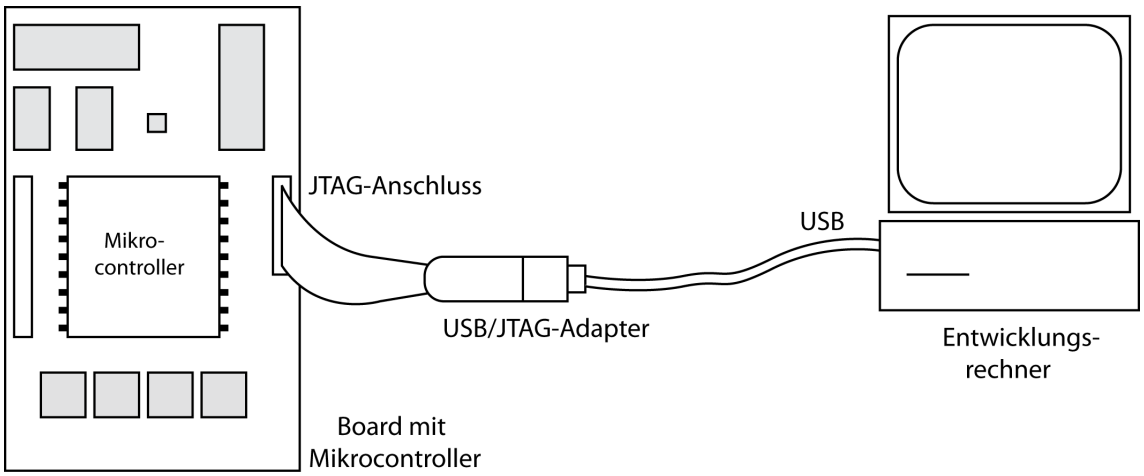


Figure 5.1: Typische Entwicklungsumgebung für einen Mikrocontroller.

Auf Entwicklungsrechner (meist ein PC) läuft eine Integrated Development Environment (IDE). Diese enthält einen Crosscompiler der Code für das Zielsystem erzeugt.

IDEs erhält man z.B. von den Prozessorherstellern oder von IAR Systems, Keil, Atol-

lic u.a.m. Es gibt für viele Controller auch kostenlose Compiler, wie zum Beispiel den GNU-Compiler GCC und es gibt limitierte IDE-Versionen kostenlos.

Die Embedded Workbench von IAR

Eine sehr gute und bewährte Entwicklungsumgebung ist die "'Embedded Workbench"' von IAR. Sie ist als Volversion, als zeitlich begrenzte Testversion und als Codegrößen-begrenzte "'Kickstart"'-Umgebung verfügbar.

- Zur Übersetzung muss immer ein Projekt angelegt werden.
- Es gibt einen Workspace, der mehrere Projekte enthalten kann.
- Bei einem neuen Projekt muss zunächst der aktuelle Prozessortyp eingestellt werden und ausgewählt werden, ob Simulation oder echtes Target.
- Wenn das Projekt fehlerfrei übersetzt wird kann es bequem auf via USB und JTAG das Target übertragen werden.
- Im Dateifenster kann eingestellt werden, welche Dateien aktuell zum Projekt gehören sollen. (und mit übersetzt werden)

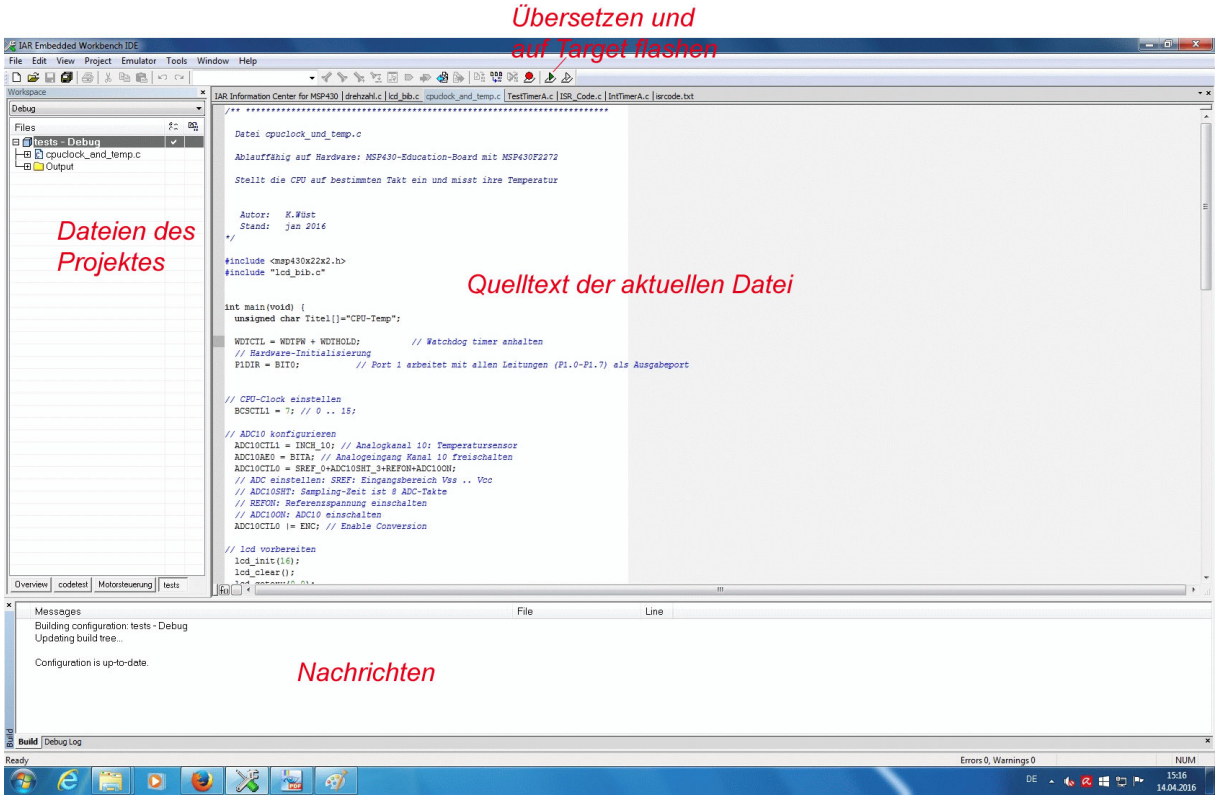


Figure 5.2: Geöffnetes Projekt in der "'Embedded Workbench"' von IAR.

Wenn das Programm auf das Target geflasht ist, wird es in den Debugger geladen und befindet sich zunächst im Halt-Zustand. (Abb. 5.3) Nun kann eine echte Ausführung auf dem Target angestoßen werden. dabei hat man die Auswahl zwischen

Einzelschritt und "'Run"' (flüssige Ausführung). Wenn es flüssig abläuft, kann man es auch wieder stoppen. Danach kann man:

- *alle* Register betrachten und verändern,
- alle Bereiche des Speichers betrachten und verändern,
- Haltepunkte (Breakpoints) setzen oder löschen,
- das Programm in den Startzustand versetzen,
- im Disassembly-Fenster den erzeugten Assemblercode und den erzeugten Maschinencode sehen.

Wenn man das Programm ändern will verlässt man den Debug-Modus wieder und übersetzt danach neu.

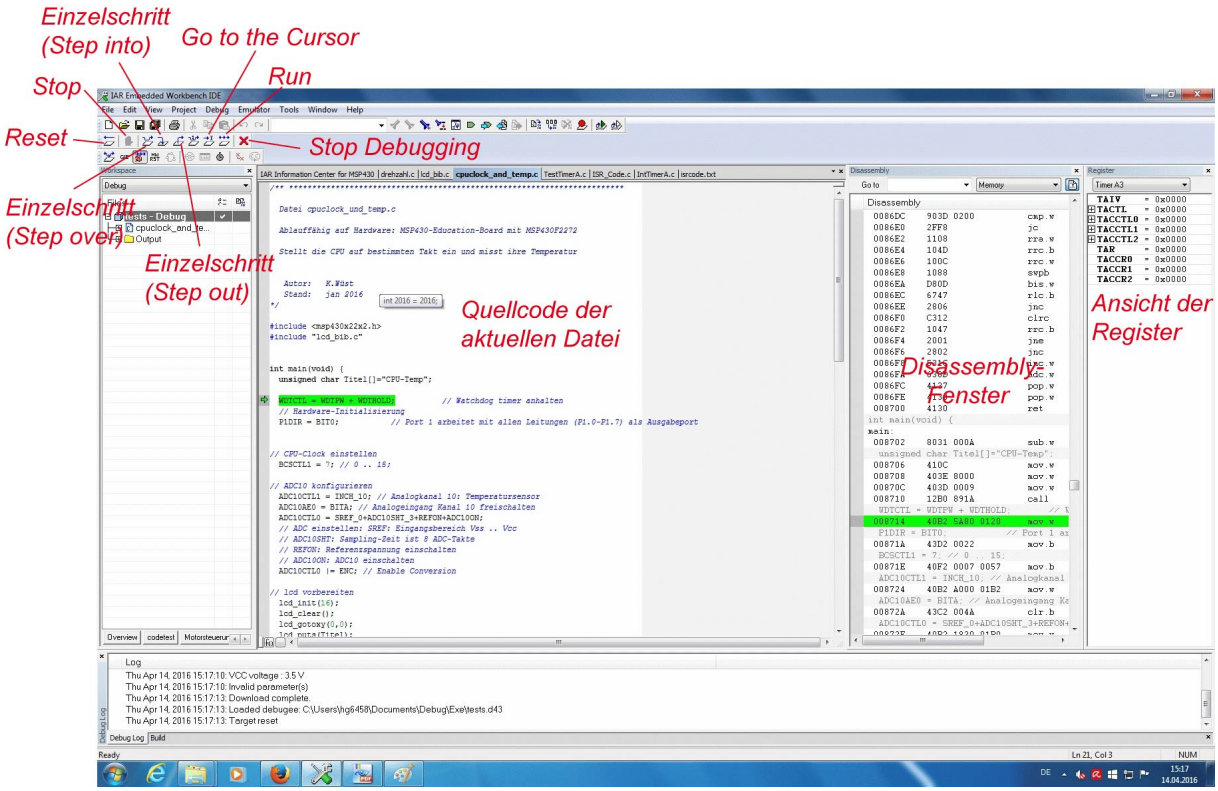


Figure 5.3: Programm im Debug-Modus. (Abb. von der "'Embedded Workbench"' von IAR.)

5.2 Programmstruktur (Teil I)

Die Struktur der Programme kann sehr verschieden sein. Sie hängt stark von den jeweiligen Bedingungen und Anforderungen ab, insbesondere von

- Der Größe des Programmspeichers (Reicht der Platz für ein Betriebssystem?)
- Dem Umfang der zu leistenden Funktionalität (Wie viele Funktionen, externe Ereignisse, Busprotokolle, verschachtelte Interrupts und Echtzeitanforderungen liegen vor?)
- Der Komplexität der CPU (Gibt es Speicherschutz-Mechanismen, Privilegierungsstufen u.ä.?)

Die einfachste Programmstruktur: Round-Robin

Round-Robin-Struktur kann man übersetzen als Ringstruktur und so funktioniert es auch: In einer großen Endlosschleife werden alle Systemaufgaben wahrgenommen. Das Round-Robin-Programm initialisiert zunächst die Hardware des Controllers und des Boards und geht dann in eine Endlosschleife; ein Betriebssystem gibt es nicht. Die Programmschema:

```
int main(void)
{
  < Hardware initialisieren >

  while(1) {
    if (<Baustein 1 braucht Service) {    // z.B. Tastatur abfragen
      < Baustein 1 bedienen >             // auf Taste reagieren
      }
    if (<Baustein 2 braucht Service) {    // z.B. Sensor abfragen
      < Baustein 2 bedienen >             // Ergebnisse berechnen
      }
    if (<Baustein 3 braucht Service) {    // z.B. Anzeige noch aktuell?
      < Baustein 3 bedienen >             // Anzeige aktualisieren
      }

    // weitere Bausteine ...
  }  // end while
}
```

Vor und Nachteile von Round-Robin sind:

- Struktur ist wunderbar einfach und ergibt in vielen Fällen eine effiziente und schnelle Lösung.
- Funtioniert gut, wenn die Anwendung nicht zeitkritisch und der Prozessor nicht ausgelastet ist; daher sehr beliebt.
- Dauer der Systemreaktion (Latenzzeit) nicht vorhersagbar (Programm ist irgendwo in der Schleife) Jeder Baustein wird erst dann wieder beachtet, wenn

er turnusmäßig an der Reihe ist.

- Feste Bearbeitungsreihenfolge (keinen Vorrang für dringende Aufgaben)
- Schlechte Energieeffizienz: Der Prozessor bearbeitet endlos alle Services in Höchstgeschwindigkeit ohne jemals in einen Low-Power-Mode zu gehen

5.3 Entwicklung eigener Programme

Phasen der SW-Erstellung

Planung Wie soll die Aufgabe gelöst werden? (Siehe unten)

Entwurf Wie soll das Programm strukturiert sein, welche Funktionen werden gebraucht?

Codierung Umsetzung des Entwurfs in ein C-Programm.

Test und Verifikation Erkennung interessanter Fälle und Situationen, Planung der Tests, Test einzelner Funktionen oder des ganzen Programms, Testprotokoll. Ist die Bedienbarkeit gut, kann die Anzeige gut abgelesen werden?

Dokumentation Dokumentation des Programms, gerne zum Beispiel mit Doxygen

Planung im Detail

- Welche Peripheriekomponenten und welche Softwarekomponenten (Bibliotheken) werden gebraucht.
- Wie sind die Abhängigkeiten der Ereignisse?
- Wie sollen die Peripheriekomponenten betrieben werden? Wie müssen diese Baugruppen konfiguriert werden?
- Werden Interrupts gebraucht?
- Welche Leitungen von welchen Hardwarebaugruppen müssen benutzt werden? Gibt es Konfklikte?
- Wie können die Leitungen benutzt werden, was bedeutet jeweils HIGH/LOW
- Wie gehe ich mit dem Watchdog Timer um? (Benutzen oder Abschalten?)
- Wie kann man das Programm testen, was sind interessante Situationen?

Dazu auch Aufzeichnungen auf Papier machen! Auch UML bietet Unterstützung, z. B. durch Zustandsdiagramme, Sequenzdiagramme und Kollaborationsdiagramme

Aufbau des Quellcodes

- Benutzen Sie Einrückungen
- Fügen Sie genug Kommentare ein
- Bündeln Sie Hilfsroutinen, die in mehreren Sourcefiles gebraucht werden, in

eigenen Bibliotheks-Dateien, die dann per include-Anweisung bei der Übersetzung eingefügt werden

- Vermeiden Sie Vervielfältigung mit "'Copy und Paste"' (Warum?)

PRAKTIKUMSAUFGABEN NACH DIESEM ABSCHNITT

1. *Aufgabe 1 Eine Leuchtdiode blinken lassen*
2. *Aufgabe 2 Blinkprogramm ändern: Zwei Leuchtdioden leuchten abwechselnd*

6 Software-Entwicklung für Mikroprozessoren (Teil III)

6.1 Allgemeines über Bitoperationen

Wahrheitstabelle der Bitoperatoren:

A	B	NOT A	A AND B	A OR B	A XOR B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

AND und OR sind binäre Operatoren, NOT ist ein unärer Operator. In der Programmiersprache C sind diese Operatoren alle verfügbar. Dazu kommen die beiden unären Schiebeoperationen "'‘Schieben nach links'" und "'‘Schieben nach rechts'"

Binäre Operatoren

&	AND	(UND)
|	OR	(ODER)
^	XOR	(Exclusive Oder)

Unäre Operatoren

~	NOT	(NICHT, invertieren)
<<	Shift Left	(Schieben nach links)
>>	Shift Right	(Schieben nach rechts)

In C wirken die Bitoperatoren nicht auf eine einzelne logische Variable, sondern auf jedes einzelne Bit einer C-Variablen:

Alle Operatoren funktionieren auch in Kombination mit dem Zuweisungsoperator =, also `&=`, `|=`, `^=`, `<<=`, `>>=`.

Beispiel 1 Schieben nach links

```
unsigned char var1=0x06;  // var1 ist 00000110b = 6d
var1 <<= 4;          // var1 ist jetzt 01100000b also 96d
```

Das Schieben nach links entspricht einer Multiplikation; Schieben um 4 Bit entspricht einer Multiplikation mit 16.

Beispiel 2 Schieben nach rechts

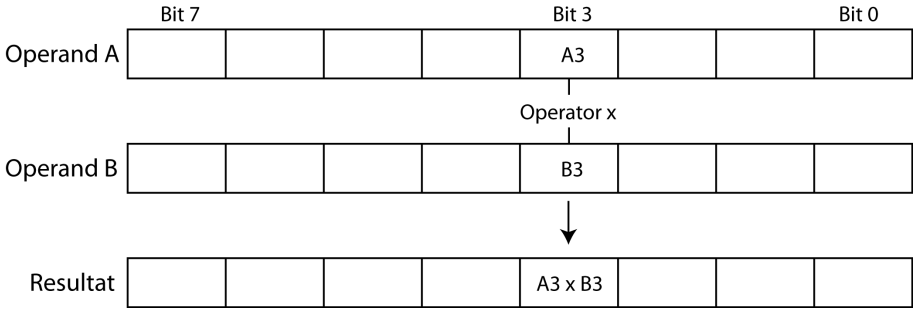


Figure 6.1: Ein logischer Verknüpfungsbefehl in einer Hochsprache arbeitet bitweise. (Nur die Verknüpfung von Bit3 dargestellt, alle anderen Bits ebenso)

```
unsigned char var1=0x06;  // var1 ist 00000110b = 6d
var1 >>= 1;          // var1 ist jetzt 00000011b also 3d
```

Das Schieben nach rechts entspricht einer Division; Schieben um 1 Bit entspricht einer Division durch 2.

Beispiel 3 Invertieren

```
unsigned char var1=0x06;  // var1 ist 00000110b = 6d
var1 ~= var1;           // var1 ist jetzt 11111001b also 249d
```

Beim Invertieren wird jedes einzelne Bit negiert.

Beispiel 4 Bitweise logisches UND

```
unsigned char var1=0x6C; // var1 ist 01101100b
unsigned char var2=0x39; // var2 ist 00111001b
unsigned char var3;
var3 = Var1 & Var2;      // var3 ist 00101000b
```

Beispiel 5 Bitweise logisches ODER

```
unsigned char var1=0x6C; // var1 ist 01101100b
unsigned char var2=0x39; // var2 ist 00111001b
unsigned char var3;
var3 = Var1 | Var2;      // var3 ist 01111101b
```

Beispiel 6 Bitweise logisches exclusives ODER (XOR)

```
unsigned char var1=0x6C; // var1 ist 01101100b
unsigned char var2=0x39; // var2 ist 00111001b
unsigned char var3;
var3 = Var1 ^ Var2;      // var3 ist 01010101b
```

Die binären bitweisen Operationen (Bsp. 4–6) ergeben wenig Sinn, wenn die Variablen als Zahlen interpretiert werden. Sie sind aber sehr sinnvoll, wenn es sich um Steuerungsmasken für Mikrocontroller handelt. Dies wird im nächsten Abschnitt gezeigt. Vor allem sucht man Wege, um einzelne Bits zu ändern und den Rest der Variable unverändert zu lassen.

Übung: 6.1 Bitmaske laden

Register Regxy soll so eingestellt werden, dass die Bits 0,2,3-7, 13 und 15 gesetzt sind und die anderen Bits gelöscht. Welcher Befehl in C bewirkt all das?

Übung: 6.2 Bit setzen

Nun soll in Register Regxy zusätzlich Bit 11 gesetzt werden. Wie lautet der C-Befehl, der das ausführt?

Übung: 6.3 Bit löschen

Nun soll in Register Regxy Bit 5 gelöscht werden. Wie lautet der C-Befehl, der das ausführt?

Übung: 6.4 Bit umschalten (toggeln)

Nun soll in Register Regxy Bit 10 umgeschaltet werden. Falls es 1 ist soll es nachher 0 sein, falls es 0 ist soll es nachher 1 sein. Wie lautet der C-Befehl, der das ausführt?

Übung: 6.5 bit abfragen

Das Statusregister Regstat liefert beim auslesen den Wert 1072. Schreiben Sie einen kurzen Abschnitt in C, der feststellt, ob Bit 4 gesetzt ist!

6.2 Bitoperationen in der MP-Programmierung

Steuerung der Hardware durch Zugriff auf die Peripherie-Register

Die Peripheriekomponenten eines Mikrocontrollers werden durch Schreiben und Lesen Ihrer Steuer- und Statusregister "'bedient"'. Dabei werden oft Bitoperationen gebraucht. Hier gilt folgende Faustregel:

- Beim ersten Beschreiben eines Konfigurationsregisters vollständig initialisieren und dazu (mit Überlegung!) das komplette Register beschreiben; das geschieht mit dem Zuweisungoperator "'="' benutzen.
- Beim späteren Ändern einzelner Bits im Konfigurationsregister sollte man Bits setzen mit der ODER-Operation ("'|="') und Bits löschen mit der UND-Operation ("'&="'); so bleiben die anderen, schon gesetzten, Bits unverändert.

Ein einführendes abschreckendes Beispiel

Bei einem MSP430F2272 soll im Steuerregister von Timer_A folgendes eingetragen werden: (Siehe dazu auch Abschn.8.3)

- Eingangstakt ist ACLK
- Der Eingangsteiler soll diesen Takt durch 8 teilen
- Betriebsart ist der Continuous Mode
- Der Zähler soll auf Null gesetzt werden.
- Der Zähler soll keine Interrupt auslösen

Datenblatt sagt aus, das Steuerregister von Timer_A (TACTL) hat die Hardwareadresse 0162h. Der User's Guide verrät uns die Bedeutung der Bits und Bitfelder in diesem Register. (Abb. 6.2)

Um die oben genannten Einstellungen vorzunehmen müssen also die Bitfelder wie folgt beschrieben werden:

Timer A Source Select (TASSEL)	Bits 9–8	01
Input Divider (ID)	Bits 7–6	11
Mode Control (MC)	Bits 5–4	10
Timer A Clear (TACLR)	Bit 2	1
Timer A Interrupt Enable (TAIE)	Bit 1	0

Wenn man alle Bits in ein 16-Bit-Wort einfügt und die restlichen Bits Null setzt, erhält man die Bitmaske 0000 0001 1110 0100b. Das Bit TAIE wird damit – wie gewünscht – auf Null gesetzt. Wenn wir diesen Wert hexadezimal ausdrücken, ergibt sich 01E4h. Unser C-Code für diese Aufgabe könnte also sein:

TACTL, Timer_A Control Register

15	14	13	12	11	10	9	8
Unused						TASSELx	
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)

7	6	5	4	3	2	1	0
IDx		MCx		Unused	TACLR	TAIE	TAIFG
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	w–(0)	rw–(0)	rw–(0)

Unused	Bits 15-10	Unused
TASSELx	Bits 9-8	Timer_A clock source select 00 TACLK 01 ACLK 10 SMCLK 11 INCLK
IDx	Bits 7-6	Input divider. These bits select the divider for the input clock. 00 /1 01 /2 10 /4 11 /8
MCx	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power. 00 Stop mode: the timer is halted. 01 Up mode: the timer counts up to TACCR0. 10 Continuous mode: the timer counts up to 0FFFFh. 11 Up/down mode: the timer counts up to TACCR0 then down to 0000h.
Unused	Bit 3	Unused
TACLR	Bit 2	Timer_A clear. Setting this bit resets TAR, the clock divider, and the count direction. The TACLR bit is automatically reset and is always read as zero.
TAIE	Bit 1	Timer_A interrupt enable. This bit enables the TAIFG interrupt request. 0 Interrupt disabled 1 Interrupt enabled
TAIFG	Bit 0	Timer_A interrupt flag 0 No interrupt pending 1 Interrupt pending

Figure 6.2: Bedeutung der Steuerbits im Register TACTL des Timer_A. Aus dem MSP430 Family User's Guide mit freundlicher Genehmigung von Texas Instruments.

```
int *p;                 // Zeiger anlegen

 p= (int*)(0x0162);     // Adresse des TACTL auf Zeiger schreiben
*p = 0x01E4;            // Bitmaske einschreiben
```

Dieser Code würde funktionieren, hat aber eine Reihe von Nachteilen.

- Zieladresse schwer lesbar: Welches Register liegt unter Adresse 162h? (Datenblatt nötig!)
- Bitmaske schwer verständlich: Was bedeutet 0x01E4? (Datenblatt wieder nötig!)
- Fehler können leicht passieren
- Wenn man auf einen anderen Controller wechselt: Ganzes Programm muss

mühsam durchgeschaut und manuell geändert werden.

Besserer Weg: Benutzung der vom Hersteller mitgelieferten Header-Dateien, siehe unten.

Die Benutzung Symbolische Konstanten und Header-Dateien

Mit der Entwicklungsumgebung werden so genannte Header-Files (xxx.h) ausgeliefert. In diesen Header-Files findet

- symbolische Konstanten für die Adressen der Register
- symbolische Konstanten für die Steuerbits
- Vordefinierte Bitkonstanten

Diese Konstanten machen das Programm wesentlich leichter lesbar und wartbar. In unserem Fall ist die Datei "'msp430x22x2.h"' die richtige. Sie ist zu finden im Pfad IAR-Systems/Embedded Workbench/430/inc. Beginnen wir mit den vordefinierten Bitkonstanten, die schon vieles erleichtern:

```
#define BIT0    (0x0001)
#define BIT1    (0x0002)
#define BIT2    (0x0004)
#define BIT3    (0x0008)
#define BIT4    (0x0010)
#define BIT5    (0x0020)
#define BIT6    (0x0040)
#define BIT7    (0x0080)
#define BIT8    (0x0100)
#define BIT9    (0x0200)
#define BITA    (0x0400)
#define BITB    (0x0800)
#define BITC    (0x1000)
#define BITD    (0x2000)
#define BITE    (0x4000)
#define BITF    (0x8000)
```

Übung: 6.6 Bitkonstanten verwenden

Schreiben Sie die Lösungen der Übungen in Abschnitt 6.1 unter Benutzung dieser Bitkonstanten neu auf.

Weiter zu den symbolischen Adress-Konstanten und Steuerbit-Konstanten; diese Konstanten erleichtern den Zugriff auf die Hardwaregruppen enorm. Wir geben hier einen kleinen Auszug wieder, der den Analog/Digital-Converter ADC10 betrifft:

```
/************************************************************
* ADC10
************************************************************/
#define __MSP430_HAS_ADC10__  /* Definition to show that Module is available */

#define ADC10DTC0_          (0x0048)    /* ADC10 Data Transfer Control 0 */
DEFC(   ADC10DTC0         , ADC10DTC0_)
#define ADC10DTC1_          (0x0049)    /* ADC10 Data Transfer Control 1 */
DEFC(   ADC10DTC1         , ADC10DTC1_)
#define ADC10AE0_           (0x004A)    /* ADC10 Analog Enable 0 */
DEFC(   ADC10AE0          , ADC10AE0_)
#define ADC10AE1_           (0x004B)    /* ADC10 Analog Enable 1 */
DEFC(   ADC10AE1          , ADC10AE1_)

#define ADC10CTL0_          (0x01B0)    /* ADC10 Control 0 */
DEFW(   ADC10CTL0         , ADC10CTL0_)
#define ADC10CTL1_          (0x01B2)    /* ADC10 Control 1 */
DEFW(   ADC10CTL1         , ADC10CTL1_)
#define ADC10MEM_           (0x01B4)    /* ADC10 Memory */
DEFW(   ADC10MEM          , ADC10MEM_)
#define ADC10SA_            (0x01BC)    /* ADC10 Data Transfer Start Address */
```

```
DEFW(   ADC10SA          , ADC10SA_)

/* ADC10CTL0 */
#define ADC10SC             (0x001)     /* ADC10 Start Conversion */
#define ENC                 (0x002)     /* ADC10 Enable Conversion */
#define ADC10IFG            (0x004)     /* ADC10 Interrupt Flag */
#define ADC10IE             (0x008)     /* ADC10 Interrupt Enable */
#define ADC10ON             (0x010)     /* ADC10 On/Enable */
#define REFON               (0x020)     /* ADC10 Reference on */
#define REF2_5V             (0x040)     /* ADC10 Ref 0:1.5V / 1:2.5V */
#define MSC                 (0x080)     /* ADC10 Multiple SampleConversion */
#define REFBURST            (0x100)     /* ADC10 Reference Burst Mode */
#define REFOUT              (0x200)     /* ADC10 Enable output of Ref. */
#define ADC10SR             (0x400)     /* ADC10 Sampling Rate 0:200ksps /1:50ksps */
#define ADC10SHT0           (0x800)     /* ADC10 Sample Hold Select 0 */
#define ADC10SHT1           (0x1000)    /* ADC10 Sample Hold Select 1 */
#define SREF0               (0x2000)    /* ADC10 Reference Select 0 */
#define SREF1               (0x4000)    /* ADC10 Reference Select 1 */
#define SREF3               (0x8000)    /* ADC10 Reference Select 2 */
#define ADC10SHT_0          (0*0x800u)   /* 4 x ADC10CLKs */
#define ADC10SHT_1          (1*0x800u)   /* 8 x ADC10CLKs */
#define ADC10SHT_2          (2*0x800u)   /* 16 x ADC10CLKs */
#define ADC10SHT_3          (3*0x800u)   /* 64 x ADC10CLKs */

#define SREF_0              (0*0x2000u)  /* VR+ = AVCC and VR- = AVSS */
#define SREF_1              (1*0x2000u)  /* VR+ = VREF+ and VR- = AVSS */
#define SREF_2              (2*0x2000u)  /* VR+ = VEREF+ and VR- = AVSS */
#define SREF_3              (3*0x2000u)  /* VR+ = VEREF+ and VR- = AVSS */
#define SREF_4              (4*0x2000u)  /* VR+ = AVCC and VR- = VREF-/VEREF- */
#define SREF_5              (5*0x2000u)  /* VR+ = VREF+ and VR- = VREF-/VEREF- */
#define SREF_6              (6*0x2000u)  /* VR+ = VEREF+ and VR- = VREF-/VEREF- */
#define SREF_7              (7*0x2000u)  /* VR+ = VEREF+ and VR- = VREF-/VEREF- */
```

Zurück zum Beispiel der Konfiguration von Timer A mit dem (nicht empfehlenswerten) Code

```
int *p;                 // Zeiger anlegen

 p= (int*)(0x0162);     // Adresse des TACTL auf Zeiger schreiben
*p = 0x01E4;            // Bitmaske einschreiben
```

In der vom Hersteller mitgelieferten Header-Datei finden sich folgende Konstanten:

```
#define TASSEL_1    (0x0100)    /* Timer A clock source select: 1 - ACLK  */
#define ID_3        (0x00C0)    /* Timer A input divider: 3 - /8 */
#define MC_2        (0x0020)    /* Timer A mode control: 2 - Continuous up */
#define TACLR       (0x0004)    /* Timer A counter clear */
```

Diese Konstanten können einfach durch bitweises ODER verknüpft werden zu einer 16-Bit-Konstante, die alle diese Einstellungen enthält:

```
#include <msp430x22x2.h>
```

```
TACTL = TASSEL_1 | ID_3 | MC_2 | TACLR
```

Code ist jetzt weniger fehleranfällig und leichter zu verstehen. Bei Wechsel auf eine andere Hardwareplattform lädt man die neue Headerdatei dazu und mit etwas Glück braucht man seinen Code nicht zu ändern.

Statt einer Veroderung der Bitkonstanten können diese hier auch addiert werden, also:

```
TACTL = TASSEL_1 + ID_3 + MC_2 + TACLR
```

Übung: 6.7 Addieren und Verodern

a) Beweisen Sie die letzte Behauptung am Beispiel, indem Sie zeigen, dass die Addition zum gleichen Ergebnis führt!
b)Warum geht das hier, obwohl im Allgemeinen Veroderung und Addition zu verschiedenen Ergebnissen führen?

Im Header-File sind auch die Hardware-Adressen der Register definiert, z.B. für ADC10CTL0 der Wert 0x01B0. → `Datasheet 2272 S.21`

Typische Anwendungen von Bitoperationen

Bits setzen Das Bit wird in den Zustand 1 gebracht, egal welchen Zustand es vorher hatte. Geht einfach mit der ODER-1-Verknüpfung

Bits löschen Das Bit wird in den Zustand 0 gebracht, egal welchen Zustand es vorher hatte. Geht einfach mit der UND-0-Verknüpfung

Bits umschalten (toggeln) Der zustand des Bits wird geändert, egal welchen Zustand es vorher hatte. Geht einfach mit der XOR-1-Verknüpfung

Beispiel 1 Im Ausgaberegister von Port 4 sollen die Bits 0 und 4 auf 1 gesetzt

werden, egal was vorher darin stand; die restlichen Bits sollen unverändert bleiben.

```
P4OUT = P4OUT | 0x11;     // 0x11 = 00010001b
```

oder kürzer:

```
P4OUT |= 0x11;
```

oder mit Bitkonstanten:

```
P4OUT |= BIT0 | BIT4;
```

Falsch wäre der Befehl

```
P4OUT = BIT0 | BIT4;
```

Dieser Befehl setzt zwar auch die Bits 0 und 4 auf 1, er verändert aber auch die anderen Bits, die er alle auf 0 setzt.

Beispiel 2 Im Ausgaberegister von Port 1 sollen alle Bits gelöscht werden außer den Bits 1 und 2; diese beiden Bits sollen unverändert bleiben.

```
P1OUT = P1OUT & 0x06;   // 0x06 = 00000110b
```

oder kürzer:

```
P1OUT &= 0x06;
```

oder mit Bitkonstanten:

```
P1OUT &= (BIT1+BIT2);
```

Beispiel 3 Die Interrupts des MSP430 sollen abgeschaltet werden. Das Global Interrupt Enable Bit ist das Bit 3 im Statusregister; die restlichen Bits sollen unverändert bleiben.

```
SR &= 0xFFF7;
```

oder

```
SR &= (~BIT3);
```

Anmerkung: Besser benutzt man die Funktion `__disable_ints()`.

Beispiel 4 Für ein Blinklicht soll im Ausgaberegister von Port 2 das Bit 7 umgeschaltet werden; die restlichen Bits sollen unverändert bleiben.

```
P2OUT ^=  0x80;
```

oder

```
P2OUT ^=  BIT7;
```

Beispiel 5 An Port 1 sind Leuchtdioden angeschlossen, die dann leuchten, wenn der Ausgang LOW ist! Nun sollen letzten drei LEDs leuchten (Bits 5–7).

```
P1OUT = ~0x07;
```

oder

```
P1OUT = ~(BIT5|BIT6|BIT7);
```

Beispiel 6 An Port 2 sind Tasten angeschlossen, die LOW liefern, wenn die Taste gedrückt ist und HIGH, wenn sie nicht gedrückt ist! Um einfach die Tasten einzulesen genügt der Befehl

```
Tasten = ~P2IN;
```

Bitfelder

Sowohl in den Steuer- wie auch den Statusregistern wird Information verwaltet, die unterschiedliche Bitbreiten erfordert. Dabei muss ökonomisch mit dem Speicherplatz umgegangen werden. Man opfert kein ganzes 16-Bit-Register, wenn es um eine Information geht, die z.B. auf 3 Bit dargestellt werden kann.

Solche Informationen werden in Bitfelder gepackt und die Bitfelder stehen dann nebeneinander in den Registern. Es gibt Bitfelder in allen Längen. natürlich bleiben dabei manchmal Bits übrig, die sind dann "'unused"'. Ein Beispiel ist in Abb. 6.3 zu sehen. In den bisherigen Beispielen wurden schon Bitfelder benutzt, dies soll hier aber noch einmal – mit weiteren Beispielen – herausgearbeitet werden.

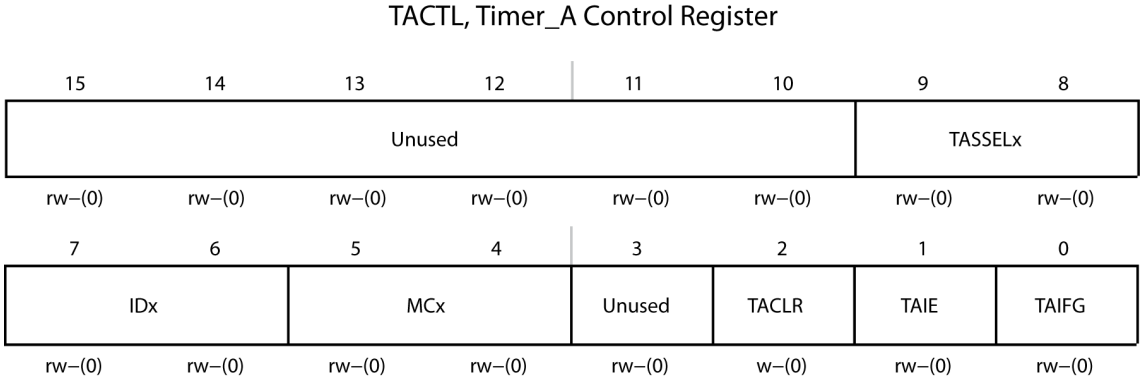


Figure 6.3: Das Control Register der Timer A-Gruppe. Mehrere Bitfelder zu 2 Bit sind erkennbar. (MC=Mode Control, ID=Input Divider und TASSEL=Timer A Source Select) (Abb. mit freundlicher Genehmigung von Texas Instruments.)

Um z.B. das Timer A-Register zu löschen (TACLR=1)und die Betriebsart auf kontinuierlich (MC=2) einzustellen muss ein Bitmuster geschrieben werden, das dem Schema 000000XXXX10X1XXb.

Ein weiteres Beispiel: das Control-Register 1 des 10-Bit-AD-Wandlers. es gibt Bitfelder mit 1, 2, 3 und 4 Bit. → MSP430Fx2xx Family User's Guide S. 609

Beim Lesen eines Bitfeldes kann in drei Schritte geschehen:

1. Register lesen
2. Gewünschtes Bitfeld "'herausmaskieren"' (alle anderen Bits auf null setzen)
3. Bitfeld ganz nach rechts schieben, damit Zahlenwert stimmt.

Die drei Schritte können in einem Befehl untergebracht werden.

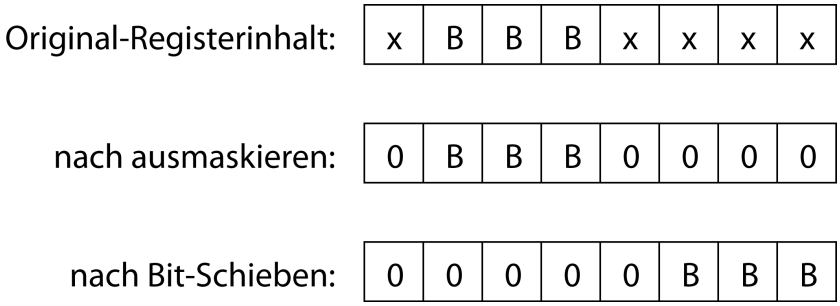


Figure 6.4: Das Lesen des Bitfeldes kann in drei Schritten geschehen. B=Bit des gesuchten Bitfeldes, x=uninteressant.

Beispiel 7 Der Mode (Betriebsart) von Timer A soll ausgelesen werden. Er steht als Bitfeld mit zwei Bit auf den Positionen 4 und 5 im TimerA-Control-Register. → MSP430Fx2xx Family User's Guide S. 410

```
TimerA_Mode = TACTL;
TimerA_Mode &= (BIT4+BIT5)
TimerA-Mode >>= 4;
```

oder in einer Zeile:

```
TimerA_Mode = ((TACTL & (BIT4+BIT5))>>4);
```

Beim Schreiben von Bitfeldern ist folgendes zu beachten:

1. Bitbreite des zu schreibenden Datums muss stimmen
2. Das zu schreibende Datum muss an der richtigen Position stehen

Beispiel 8 Der Eingangskanal des AD-Wandlers ADC10 soll auf 7 gesetzt werden. Der Eingangskanal steht auf den Bits 12–15 im Control-Register. Daher → MSP430Fx2xx Family User's Guide S. 609

```
ADC10CTL1 = (0x7000);   // Bits 12-15 enthalten die 7
```

Alternativ kann eine Variable geshiftet und dann geschrieben werden:

```
Input_Channel=7;
ADC10CTL1 = (Input_Channel<<12);   // Bits 12-15 setzen
```

Falls wir am Beginn der Konfiguration sind, beschreiben wir gleich das ganze Register mit sinnvollen Werten und benutzen die symbolischen Konstanten. Beispiel:

Beispiel 9 Es soll am Controlregister 0 des AD-Wandlers folgendes eingestellt werden: → `MSP430Fx2xx Family User's Guide S.607`

SREF = 001 (Bits 13–15)
ADC10ON = 1 (Bit 4)
SHT=01 (Bits 11–12)

Alle anderen Bitfelder sollen auf 0 bleiben, weil sie sinnvolle Voreinstellungen darstellen. Folgender Code würde in einem Schritt das ganze Controlregister setzen und einfach die Bitfelder SREF und SHT aus der Header-Datei benutzen:

```
ADC10CTL0 = SREF_1 + ADC10ON + SHT_1;  // Konfiguration des ADC-Steuerregist
```

Auch diese Formulierung ist leicht lesbar, wenig fehleranfällig und gut portierbar.

Übung: 6.8 Bitmaske

Bestimmen Sie zur Kontrolle noch einmal die Bitmaske, die man ohne die symbolischen Konstanten im letzten Befehl schreiben müsste (Ergebnis 0x2810)

PRAKTIKUMSAUFGABEN NACH DIESEM ABSCHNITT

1. *Aufgabe 3 Lauflicht*
2. *Aufgabe 4 Hallo Welt!*
3. *Aufgabe 5 Taschenrechner mit Binärausgabe*
4. *Aufgabe 6 Taschenrechner mit Textausgabe*

7 Besondere Betriebsarten

7.1 Interrupts (Unterbrechungen)

Eines der wichtigsten Konzepte in der Mikroprozessortechnik, praktisch bei allen Mikroprozessoren zu finden.

Das Problem der asynchronen Service-Anforderungen

Ein Mikroprozessor arbeitet mit zahlreichen Bausteinen auf der Systemplatine zusammen. Ein Mikrocontroller steuert Geräte. Diese Bausteine und Geräte müssen in bestimmten Situationen Daten mit dem Prozessor austauschen.

Beispiele Mikroprozessor

- Eine Taste auf der Tastatur wurde gedrückt,
- die Maus wurde bewegt,
- der Festplattencontroller sendet (zuvor angeforderte) Daten,
- auf der Netzwerkschnittstelle treffen Zeichen aus dem Netzwerk ein,
- der Zeitgeberbaustein meldet, dass die Systemzeit aktualisiert werden muss,
- der Drucker hat in seinem internen Pufferspeicher wieder Platz für weitere Daten des Druckauftrags.

Beispiele Mikrocontroller

- Eine Taster am Gerät wurde gedrückt
- Der Zähler/Zeitgeber hat einen vorgewählten Zählwert erreicht
- der Analog/Digital-Wandler hat eine Wandlung beendet, Ergebnis liegt bereit
- Über eine serielle Schnittstelle wurde ein Zeichen empfangen, es muss abgespeichert werden
- Über eine serielle Schnittstelle wurde ein Zeichen gesendet, die Schnittstelle ist wieder frei, das nächste Zeichen kann gesendet werden.

In allen diesen Fällen besteht eine *Service-Anforderung*, die der Prozessor bedienen muss. Serviceanforderungen müssen schnell bedient werden, denn:

- Interne Pufferspeicher dürfen nicht überlaufen
- Eine schnelle Systemreaktion ist gewünscht
- Bei Mikrocontrollern: Evtl. müssen Echtzeitbedingungen eingehalten werden
- Evtl. müssen Zeiten gemessen werden

Möglichkeiten zur Erfassung der Serviceanforderungen:

Polling Polling bedeutet, in einer Abfrageschleife werden alle Statusregister der Geräte und Bausteine ständig zyklisch abgefragt. Nachteil: Komplexe Programmierung, Verschwendung von Rechenzeit *Busy-Waiting*
Zum Vergleich stelle man sich vor, das Telefon hätte ein rotes Lämpchen statt einer Klingel und man müsste regelmäßig daraufschauen, um festzustellen, ob gerade jemand anruft.

Unterbrechung (Interrupt) Konzept: Das Gerät meldet selbst seine Serviceanforderung beim Prozessor. Dieser unterbricht dann seine Arbeit und kümmert sich um das Gerät. Wenn kein Interrupt anliegt, kümmert sich der Prozessor nicht um Peripheriegeräte.
In unserem Vergleich. Man vergisst das Telefon und kümmerst sich erst darum, wenn es klingelt.

Wie wird das Interrupt-Konzept technisch umgesetzt?

- Der Prozessor/Controllerkern hat einen Interrupt-Eingang (und die entsprechenden Features um einen Interrupt abzuarbeiten)
- Eine spezielle Schaltungstechnik ermöglicht es, dass mehrere Geräte/Peripheriegruppen die Interruptleitung gemeinsam benutzen
- Der Prozessor kann erkennen, von welchem Gerät der Interrupt kommt.
- Man hinterlegt Programmstücke, die den Interrupt angemessen bearbeiten: *Interrupt-Service-Routinen* (ISR) oder *Interrupt-Handler*.
- Die Adressen aller Interrupt-Service-Routinen sind in einer Tabelle, der *Interrupt-Vektoren-Tabelle*, hinterlegt, dort findet sie der Prozessor.
- Man trifft eine Regelung für folgende Situationen:
  1. Zwei Interrupts kommen gleichzeitig
  2. Während ein Interrupt bearbeitet wird, kommt ein zweiter Interrupt.
  Diese beiden Situationen werden durch eine Prioritätenregelung gelöst.
- es ist möglich, dass ein Interrupt höherer Priorität eine ISR unterbricht.

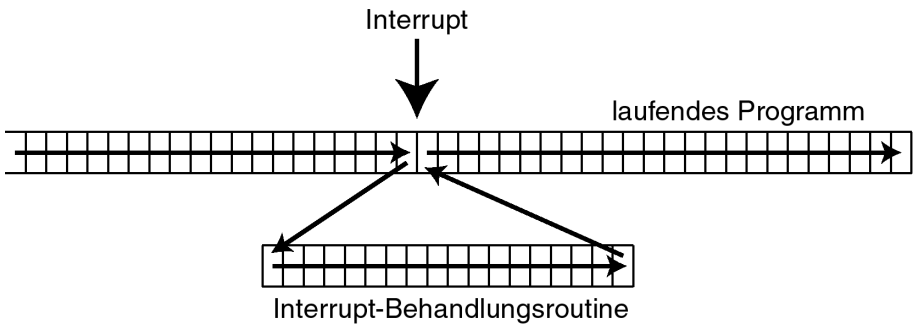


Figure 7.1: Die Unterbrechung eines Programmes durch einen Interrupt. Nach Beendigung der Interrupt-Behandlungsroutine wird das unterbrochene Programm fortgesetzt.

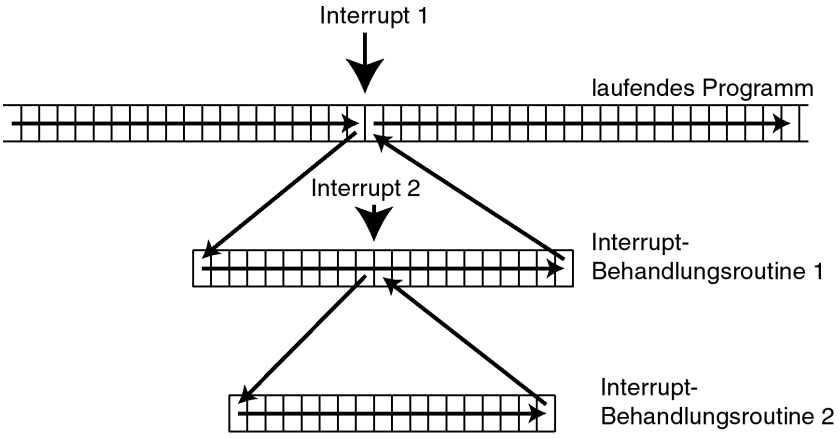


Figure 7.2: Die Unterbrechung einer Interrupt-Service-Routine durch einen weiteren Interrupt (zweistufiger Interrupt).

Ablauf eines Interrupts

- Ein Gerät löst einen Interrupt aus
- Der Prozessor nimmt den Interrupt an; er unterbricht das laufende Programm und stellt die Interruptquelle fest
- Der Prozessor speichert den momentanen Programmzähler (PC) auf den Stack
- Der Prozessor liest aus der Interrupt-Vektoren-Tabelle die Anfangsadresse der zuständigen Interrupt-Service-Routine.
- Der Prozessor verzweigt zu dieser Adresse und führt die ISR aus
- Erste Aktion in der ISR: Der Prozessor sichert seine momentane Umgebung auf den Stack
- Letzte Aktion in der ISR: Der Prozessor stellt die Umgebung wieder so her, wie vor der Unterbrechung
- Der Prozessor holt mit dem Rücksprung aus der ISR den gespeicherte Wert des PC vom Stack und setzt das unterbrochene Programm dort fort, als wäre nichts gewesen.

Freischaltung/Abschaltung von Interrupts

- Mikroprozessoren verfügen über eine Möglichkeit, den Interrupteingang intern abzuschalten; dies geschieht über ein Steuerbit ("'Flag"') das z.B. "'Global Interrupt Enable"' heißt (GIE).
- Die Interruptauslösung an den einzelnen Geräten muss in der Regel erst frei geschaltet werden.
- Bei PCs schaltet das Betriebssytem die Interruptauslösung der Geräte frei
- Bei Mikrocontrollern müssen wir selbst die Interruptauslösung der Geräte frei schalten

- Manche Prozessoren haben einen nicht abschaltbaren Interrupt-Eingang "'NMI"' (non maskable interrupt), er ist für schwerwiegende Systemfehler gedacht.

Das Abschalten aller Interrupts ist z.B. während der Konfigurierung des Interrupt-Systems angebracht.

Eigenschaften von Interrupt-Service-Routinen

Für Interrupt-Service-Routinen (ISR) gilt:

- Ein Interrupt kann jederzeit eintreten, deshalb gilt: *Eine ISR muss vor Beendigung die Umgebung exakt so wiederherstellen, wie sie zu Beginn des Interrupts war* zur Umgebung zählt: Speichervariablen, Register, Flags Wenn man in Hochsprache programmiert, kann man veranlassen, dass der Compiler diesen Code automatisch erzeugt.
- ISRs schreibt (Betriebssystem-)Entwickler selbst, er kann frei entscheiden, wie der Interrupt bearbeitet werden soll.
- ISRs müssen kurz und schnell sein, sonst kommt es zum "'Interrupt-Stau"'.
- Häufig werden die ISRs zunächst das Statusregister des betreffenden Gerätes auslesen, um mehr Informationen über die Situation zu erhalten.

7.2 Interrupt-Technik bei Mikrocontrollern

Bei Mikrocontrollern hat die Interrupt-Technik besondere Bedeutung, weil im Energiespar-Modus (Low Power Mode) die CPU abgeschaltet wird und nur durch einen Interrupt wieder erweckt werden kann!

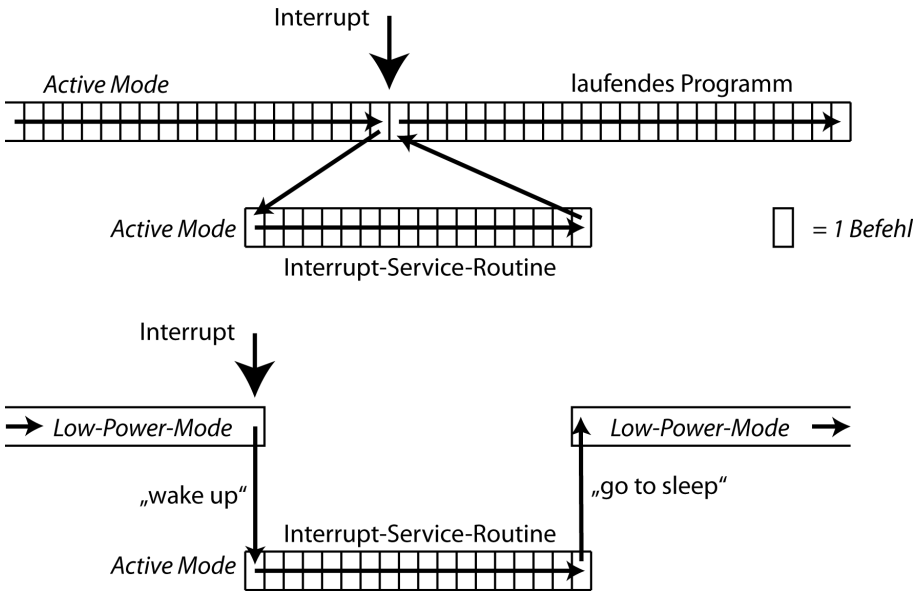


Figure 7.3: Ablauf eines Interrupts beim MSP430: Oben: Interrupt im Active Mode; Unten: Interrupt im Low-Power-Mode.

Praktisch alle Mikrocontroller haben die Möglichkeit der Interrupt-Verarbeitung. Bedeutung des Interrupt-Systems für Mikrocontroller:

- Der Mikrocontrollerkern muss viele Subsysteme ansteuern, ohne das Interruptkonzept wären viele Abfrageschleifen nötig
- Die *Echtzeitfähigkeit* von Embedded Systems beruht auf der schnellen Reaktion auf Interrupts. Nur so kann eine bestimmte Reaktionszeit garantiert werden
- Die typische Programmstruktur bei Embedded Systems schließt die Aktivierung von Interrupts ein

Table 7.1: Interrupt- und Ausnahmequellen bei Mikrocontrollern

Quelle	Ereignisse
Zähler/Zeitgeber	Compare-Ereignis, Zählerüberlauf
Serielle Schnittstelle	Zeichen empfangen, Empfangspuffer voll, Sendepuffer leer, Übertragungsfehler
Analog-Digital-Umsetzer	Umsetzung beendet
I/O Pin (externes Signal)	steigende oder fallende Flanke, HIGH-Signal
CPU	Divisionsfehler, unbekannter Opcode Daten-Ausrichtungsfehler

Die meisten modernen Mikrocontroller haben ein ausgefeiltes und flexibel konfigurierbares Interrupt-System mit beispielsweise den folgenden Eigenschaften:

- Jede Interruptquelle kann einzeln aktiviert und deaktiviert (maskiert) werden.
- Für kritische Phasen können zentral alle Interrupts deaktiviert werden.
- Für jede Interruptquelle kann ein eigener Interrupt-Handler geschrieben werden, Größe und Position im Speicher kann durch das Programm festgelegt werden.
- Für jede Interruptquelle gibt es eine definierte Priorität oder es kann eine Priorität festgelegt werden, um auch geschachtelte Interrupts zu ermöglichen.
- Das Interrupt-System kann auch im laufenden Betrieb umkonfiguriert werden.

Zusammenspiel zwischen Interrupt-Service-Routine und MC:

- Mikrocontroller speichert bei der Auslösung des Interrupts die Rücksprungadresse und (meistens) auch die Flags auf dem Stack.
- Stellt vor dem Rücksprung in das unterbrochene Programm alle Register und Flags wieder her

In der Praxis fügt die Entwicklungsumgebung automatisch den richtigen Code ein, wenn eine Funktion als "'interrupt"' gekennzeichnet wird.

Praxisbeispiel: Interrupts auf dem MSP430F2272

In der Interrupt-Vektoren-Tabelle des MSP430 ist für jeden der 32 möglichen Interrupts (nicht alle belegt) ein Interrupt-Vektor (Einsprungadresse der Interrupt-Service-Routine) hinterlegt. Jeder Interrupt-Vektor ist 16 Bit groß und belegt 2 Byte, die Tabelle umfasst also 32*2=64Byte. Sie liegt auf den Adressen 0FFc0h – 0FFFFh, also am oberen Ende des Adressraums. Die Position in der Interrupt-Vektoren-Tabelle legt gleichzeitig die Priorität fest.

Interrupt-Quelle	Adresse in Tabelle	Priorität
Power up, Externer Reset Watchdog Reset, Flash-Fehler PC ungültig	0FFFEh	31 (höchste)
NMI Oszillator-Fehler Fehlerhafter Flash-Zugriff	0FFFCh	30
Timer B3, Flag TBCCR0 CCIFG	0FFFAh	29
Timer B3, Flags TBCCR1 und TBCCR2, TBIFG	0FFF8h	28
reserviert	0FFF6h	27
Watchdog Timer, WDTIFG	0FFF4h	26
Timer A3, Flag TACCR0 CCIFG	0FFF2h	25
Timer A3, Flags TACCR1 und TACCR2, TAIFG	0FFF0h	24
USCI A0/USCI B0 Receive	0FFEEh	23
USCI A0/USCI B0 Transmit	0FFECh	22
ADC10	0FFEAh	21
reserviert	0FFE8h	20
I/O Port P2	0FFE6h	19
I/O Port P1	0FFE4h	18
reserviert	0FFE2h	17
reserviert	0FFE0h	16
(Für Steuerung Bootstrap Loader)	0FFDEh	15
Frei für Anwender	0FFDCh bis 0FFC0h	14 ... 0 (niedrigste)

Ein Interrupt läuft auf dem MSP430 nach folgendem Schema ab:

1. Eine Peripheriegruppe meldet einen Interrupt bei der CPU an.
2. Der laufende Befehl wird noch beendet; wenn der Controller im Low-Power-Mode ist, wird dieser beendet.
3. Die Hardware übermittelt auf den unteren 5 Bit des internen Datenbusses die Nummer des ausgelösten Interrupts an die CPU; diese berechnet daraus auf welche Zeile der Interrupt-Vektoren-Tabelle sie zugreifen muss. (Adresse_IVT = Int-Nr*2 + 0FFC0h)
4. Program Counter (PC) und Statusregister (SR) werden vom Controllerkern auf den Stack gespeichert. damit sind das momentane Zustandswort (einschl. LPMs) und die Rücksprungadresse gespeichert.

5. Falls mehrere Interrupts gleichzeitig anstehen, wird der mit der höchsten Priorität ausgewählt.
6. Das Statusregister wird gelöscht; dies terminiert einen eventuellen Low-Power-Mode und verhindert, dass ein weiterer Interrupt kommt.
7. Die dazu gehörende Interrupt-Service-Routine wird aufgerufen; technisch wird das bewirkt, indem der Interrupt-Vektor aus der Interrupt-Vektoren-Tabelle geholt und im PC abgespeichert wird.
8. Dadurch: Ausführung der Interrupt-Service-Routine
9. Letzter Befehl der Interrupt-Service-Routine ist IRET (statt RET); Führt eine zweifachen Zugriff auf den Stack aus:
   1. Gespeichertes Zustandswort vom Stack holen und nach Register SR schreiben,
   2. Gespeicherte Rücksprungadresse vom Stack holen und nach Register PC schreiben,

   Dadurch wird der Prozessorzustand wieder hergestellt und der Rücksprung an die Unterbrechungsstelle realisiert.
10. Die Programmausführung geht da weiter, wo der Interrupt eingetreten ist bzw. geht wieder in den Low-Power-Mode.

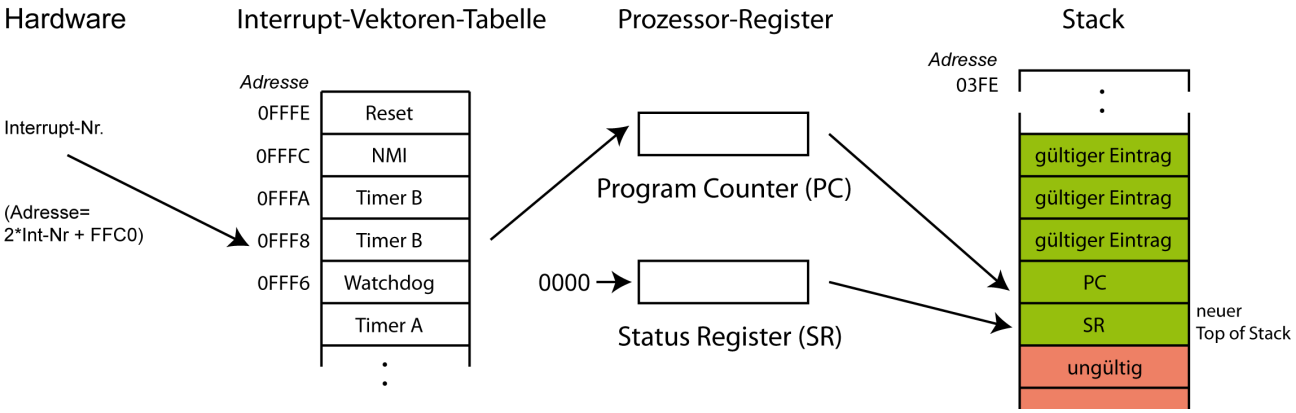


Figure 7.4: Vorgänge bei Annahme eines Interrupts beim MSP430. Die Inhalte von PC und SR werden natürlich auf den Stack gerettet *bevor* sie überschrieben werden.

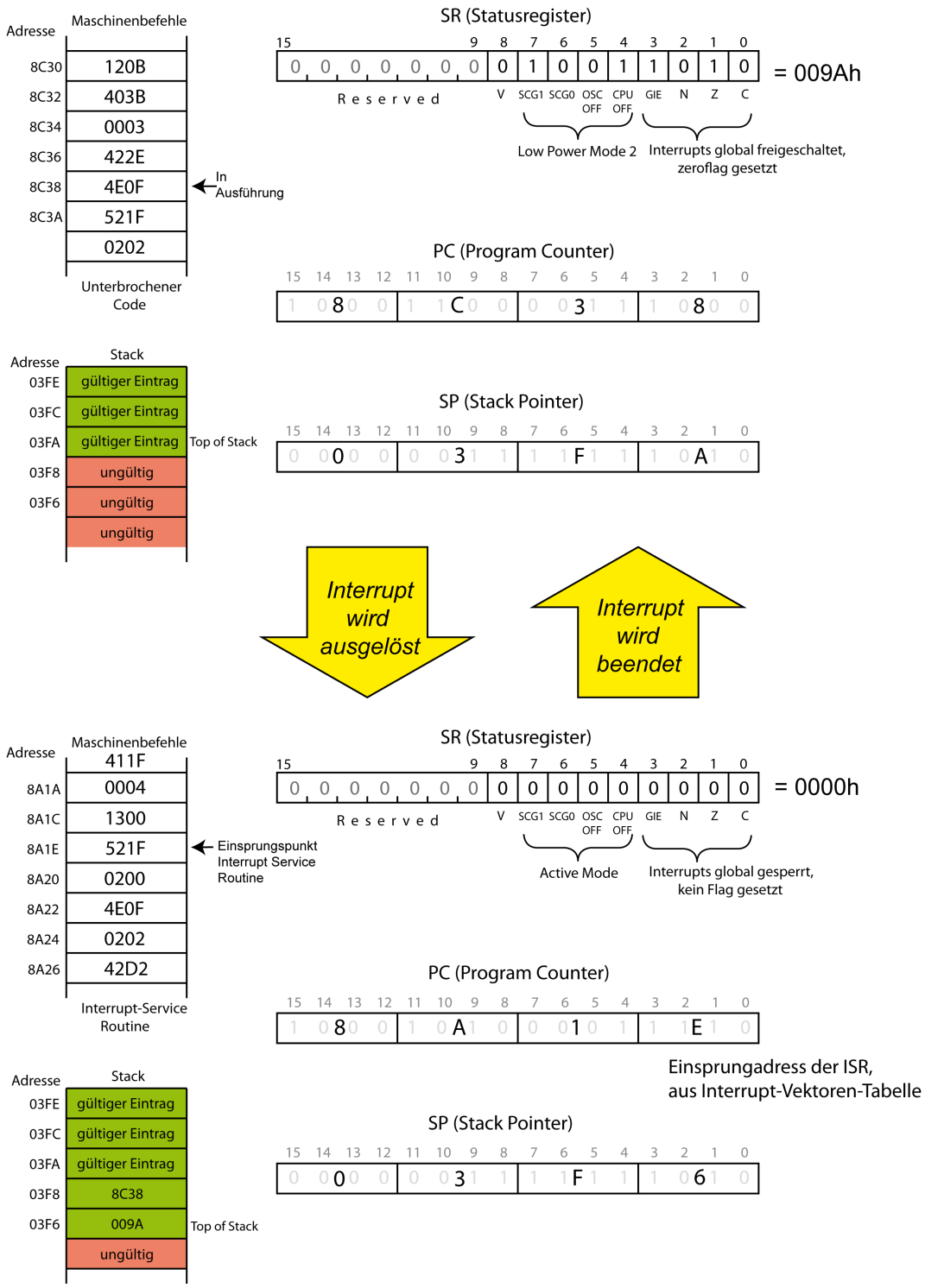


Figure 7.5: Einbindung der Interrupt-Service-Routine beim MSP430. Der in Ausführung befindliche Befehl wird noch beendet. Dann wird der PC und das SR auf den Stack gespeichert, SR wird gelöscht. Aus der Interrupt-Vektoren-Tabelle wird die Einsprungadresse der Interrupt Service Routine (ISR) in den PC geladen. Damit wird automatisch dort fortgesetzt. Die ISR endet mit dem Befehl IRET (Return from Interrupt), dieser holt zunächst das Statusregister und dann den PC vom Stack zurück. Dadurch wird der alte Zustand wieder hergestellt und an der Unterbrechungsstelle fortgesetzt. Für die Wiederherstellung der Register vor dem IRET ist der Programmierer bzw. der Compiler verantwortlich! (Codeinhalte und Codeadressen willkürlich gewählt.)

Übung: 7.1 Interrupt-Eintritt

1. Bei Eintritt eines Interrupts muss ein Low-Power-Mode automatisch beendet werden. Warum?
2. Wie wird diese Beendigung technisch ausgeführt?
3. Die Voreinstellung ist: Es können keine weiteren Interrupts eintreten, wenn man in der Interrupt-Service-Routine ist. Wie wird das technisch sichergestellt
4. Was sollte eine ISR erledigen, die aufgerufen wird, wenn in einer seriellen Schnittstelle eine Zeichen eingetroffen ist/ ein Zeichen komplett gesendet wurde?

Interrupt-Service-Routinen auf dem MSP430

Der Programmierer schreibt die Interrupt-Service-Routinen als C-Funktion. Darin legt er fest, was genau geschehen soll, falls der Interrupt eintritt. Ein Interrupt soll so schnell abgewickelt werden, dass man ihn gar nicht bemerkt. Eigenschaften von Interrupt-Service-Routinen:

- ISR sind immer vom Typ "'void"', sie können keine Werte zurück geben,
- ISR haben immer leere Parameterliste, man kann keine Parameter übergebe,
- Sie dürfen keine bleibenden Veränderungen an der Hardware (Speicher, Register) vornehmen,
- Sie sollen kompakt und kurz sein.

Programmierung der ISRvon Interrupts am MSP430:

1. Interrupt-Service-Routine als normale C-Function schreiben, mit dem Zusatz `__interrupt`, beispielsweise
   `__interrupt void Zeitnahme(...)`
2. Die Adresse dieser Function muss vom Compiler in die Interupt-Vektoren-Tabelle eingetragen werden, dazu dient der Vorsatz
   `#pragma vector=TIMERA0_VECTOR`
3. Im Hauptprogramm die Annahme von Interrupts global freischalten mit
   `__enable_interrupt();`
4. Peripheriegruppe konfigurieren und dort Interruptauslösung aktivieren.

Aussehen einer (kurzen) Interrupt-Service-Routine beim MSP430 (C-Code)

```
/* ************************************************************

 Timer A0 Interrupt Service Routine
 wird jedesmal aufgerufen, wenn Interrupt CCR0 von TimerA kommt
 Routine schaltet nur Leitung P1.0 um (toggeln)

************************************************************ */

#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A0 (void)
{
  P1OUT ^= 0x01;            // Leitung 0 von Port 1 toggeln
}
```

Die Zeile "'`#pragma vector=TIMERA0_VECTOR`"' ist eine Anweisung an den Übersetzer: Die Adresse dieser Routine ist in die Interrupt-Vektoren-Tabelle einzutragen und zwar auf dem Platz der zum Interrupt von Timer A0 gehört.

Der ergänzende Typbezeichner "'`__interrupt`"' ist auch eine Anweisung an den Übersetzer: Diese Routine als Interrupt-Service-Routine übersetzen, das heißt: Alle Register und Flags erhalten bei Verlassen der Routine wieder den anfänglichen Inhalt.

Wir betrachten Code-Erzeugung für die folgende eine kleine C-Funktion:

```
void tausche_function(void) {
    int x=2, y=3, z=4, temp;
    temp=z;
    z=y;
    y=x;
    x=temp;
}
```

Bei der Übersetzung entsteht der folgende Code:

```
tausche_function:
     int x=2, y=3, z=4, temp;
 008078    432F               mov.w   #0x2,R15
     int x=2, y=3, z=4, temp;
 00807A    403E 0003          mov.w   #0x3,R14
     int x=2, y=3, z=4, temp;
 00807E    422D               mov.w   #0x4,R13
     temp=z;
 008080    4D0C               mov.w   R13,R12
     z=y;
 008082    4E0D               mov.w   R14,R13
     y=x;
 008084    4F0E               mov.w   R15,R14
     x=temp;
 008086    4C0F               mov.w   R12,R15
 }
 008088    4130               ret
```

Register R12, R13, R14 und R15 werden einfach überschrieben und haben nach Ausführung der Funktion neue Inhalte. Der Rücksprung erfolgt mit ret (Return) Wenn man den gleichen Code mit dem Zusatz "`__interrupt`" neu übersetzt (`__interrupt void tausche_ISR(void)`) entsteht folgender Code:

```
 {
tausche_ISR:
 008042    120D               push.w  R13
 008044    120C               push.w  R12
 008046    120F               push.w  R15
 008048    120E               push.w  R14
     int x=2, y=3, z=4, temp;
 00804A    432F               mov.w   #0x2,R15
     int x=2, y=3, z=4, temp;
 00804C    403E 0003          mov.w   #0x3,R14
     int x=2, y=3, z=4, temp;
 008050    422D               mov.w   #0x4,R13
     temp=z;
```

```
008052    4D0C              mov.w   R13,R12
    z=y;
008054    4E0D              mov.w   R14,R13
    y=x;
008056    4F0E              mov.w   R15,R14
    x=temp;
008058    4C0F              mov.w   R12,R15
}
00805A    413E              pop.w   R14
00805C    413F              pop.w   R15
00805E    413C              pop.w   R12
008060    413D              pop.w   R13
008062    1300              reti
```

Man sieht, dass der Compiler einige Befehle hinzugefügt hat: Die Register werden vor der Benutzung auf den Stack gerettet (push ...) und nach Ablauf der ISR wieder hergestellt (pop ...). Der Rücksprung erfolgt mit reti (Return from Interrupt).

7.3 Ausnahmen (Exceptions)

Ähnlichkeit zu den Interrupts haben die Ausnahmen (Exceptions, auch "“Traps"'). Dabei wird die Unterbrechung nicht von einem externen Gerät, sondern vom Prozessor selbst ausgelöst. Der Grund ist eine schwerwiegende Fehlersituation, in der aus der Sicht des Prozessors das weitere Vorgehen unklar ist und durch eine ISR behandelt werden muss. Typische Fälle von Ausnahmen sind

- Divisionsfehler (Division durch Null oder zu großes Resultat)
- Unbekannter Opcode,
- Überschreitung des darstellbaren Zahlenbereiches,
- Einzelschrittbetrieb aktiviert (Debug-Betrieb),
- Feldgrenzenüberschreitung,
- Seitenfehler,
- unerlaubter Speicherzugriff,
- unberechtigter Aufruf eines privilegierten Befehls,
- Aufruf des Betriebssystems, das im privilegierten Modus arbeitet.

Die letzten vier aufgezählten Ausnahmen können nur bei Prozessoren mit Speicherverwaltung und Schutzmechanismen auftreten.

7.4 Direct Memory Access (DMA)

In einem Rechnersystem kommt es oft zur stupiden Übertragung von größeren Blöcken Daten. Man überträgt diese Aufgabe oft an einen *DMA-Controller* (Direct Memory Access, direkter Speicherzugriff). Der DMA-Controller ist auf die Übertragung von Daten spezialisiert und entlastet die CPU von dieser stumpfsinnigen Aufgabe. Nehmen wir als Beispiel die Übertragung eines Datenblock von einem Festplattencontroller zum Speicher:

Die CPU muss für die Übertragung jedes Datenwortes (Anzahl Bit = Busbreite) zwei Schritte ausführen:

1. Datum vom Controller in ein Prozessorregister übertragen (laden)
2. Datum vom Prozessorregister weiter zum Speicher übertragen (speichern).

Der DMA-Controller spart den Umweg über den Prozessor; er schaltet den Festplattencontroller auf Ausgabe und den Speicher auf Eingabe lässt in einem schritt die Daten direkt von der Quelle zum Ziel fließen; dadurch erledigt er beides in einem Schritt.

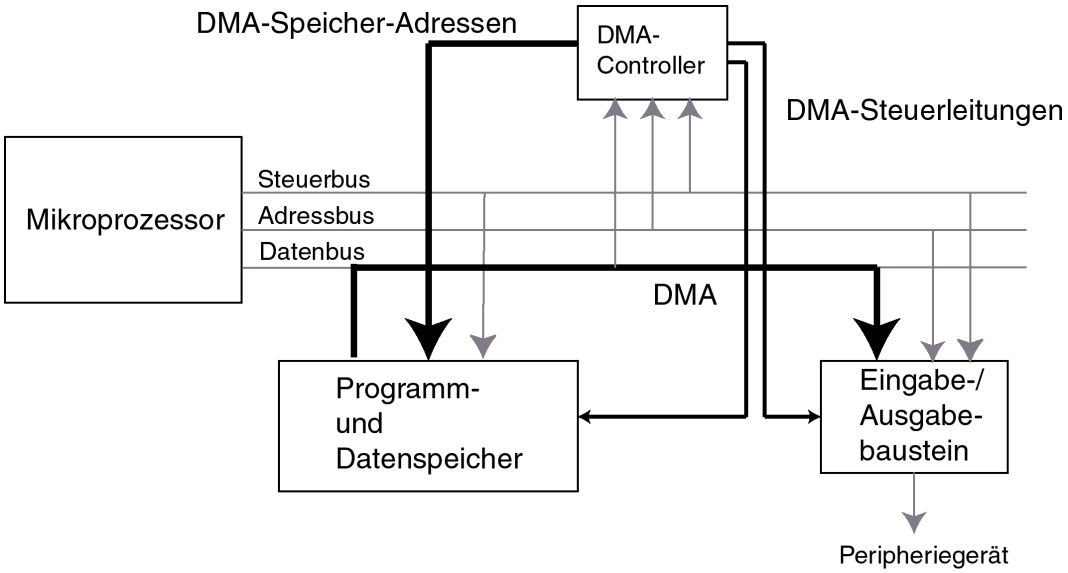


Figure 7.6: Beim Direct Memory Access gelangen die Daten ohne Umweg direkt von der Quelle zum Ziel. Es wird kein Umweg über den Prozessor gemacht.

Der DMA-Controller benutzt den Systembus, er ist also ein zweiter Busmaster. Während des DMA-Transfer ist der Bus belegt. DMA-Controller sind komplexe Bausteine mit vielen Registern.

DMA bei Mikrocontrollern

Hier wird DMA z.B. eingesetzt um bei regelmäßiger Erfassung von Messwerten diese automatisch (ohne Aktivität der CPU) auf vorbestimmte Speicherplätze abzulegen. Dort sind sie also immer aktuell können bei Bedarf durch die CPU gelesen

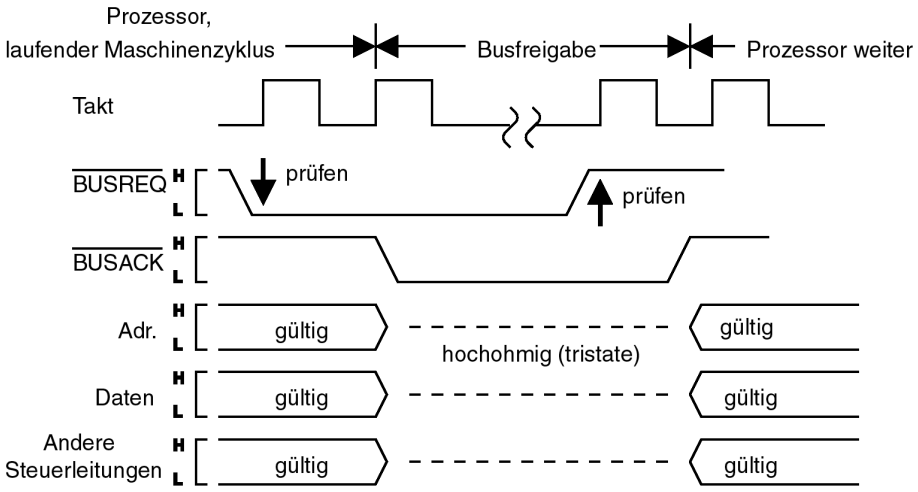


Figure 7.7: Die Busübergabe an den DMA-Controller. Dargestellt sind die Signale aus Sicht des Prozessors.

und ausgewertet werden.

Übung: 7.2 Testfragen

1. Warum müssen Service-Anforderungen schnell bedient werden?
2. Nennen Sie die wichtigsten Eigenschaften von Interrupt-Service-Routinen!
3. Was sind die Hauptaufgaben eines Interrupt-Controllers?
4. Wie kann man sicherstellen, dass ein Gerät zeitweilig keinen Interrupt auslöst?
5. Warum kann bei DMA im gleichen Buszyklus ein Speicherplatz und ein Peripheriegerät angesprochen werden?

8 Mikrocontroller: Die Zähler-/Zeitgebereinheit

8.1 Funktionsweise

Zähler/Zeitgeber-Bausteine sind Typische und sehr wichtige Peripheriegruppen, besitzt fast jeder Mikrocontroller.

Nutzen: Schon bei einfacher Impulszählung offensichtlich:

Ohne Zählerbaustein CPU muss betreffenden Eingang in einer Programmschleife ständig abfragen (pollen) und bei jeder zweiten Flanke den Wert einer Speichervariablen inkrementieren.

Mit Zählerbaustein Zählerbaustein wird konfiguriert und arbeitet danach selbständig, CPU ist frei!

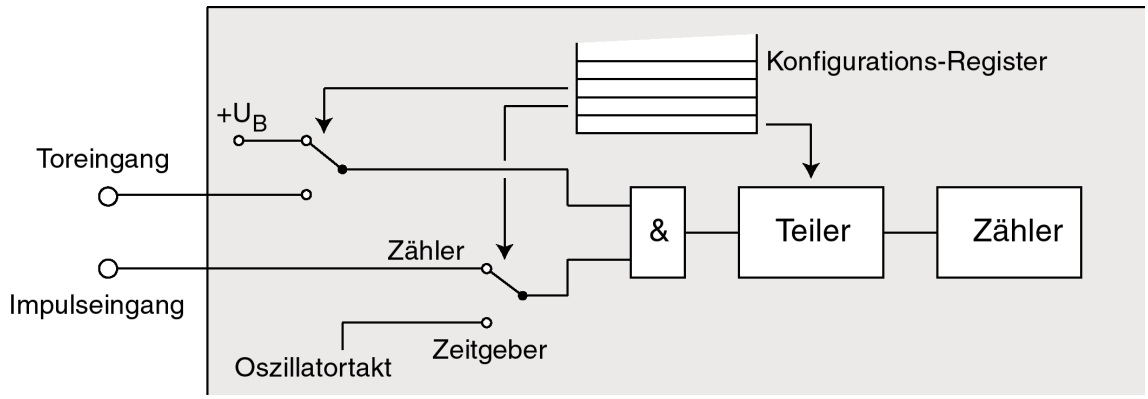


Figure 8.1: Eine Zähler/Zeitgeber-Einheit. Die Aktivierung des Toreinganges, die Umschaltung zwischen Zähler- und Zeitgeberfunktion und der Teiler werden jeweils durch Konfigurations-Register gesteuert.

Das Kernstück des Zähler/Zeitgeberbausteins ist ein Zähler, der durch eingehende Impulse inkrementiert oder dekrementiert wird (Abb. 8.1). Im *Zählerbetrieb* (Counter) Zwei grundsätzliche Betriebsarten:

Zählerbetrieb (Counter) Impulse kommen über einen Anschlussstift von aussen in den Mikrocontroller und werden einfach gezählt. Vorteiler möglich.

Zeitgeberbetrieb (Timer) Impulse werden durch Herunterteilen des internen Oszillatortaktes gewonnen. Da der Oszillatortakt bekannt ist, korrespondiert der Zählwert mit der vergangenen Zeit. Exakte Zeitmessungen sind möglich!

Zusatzausstattung: *Torsteuerung*, Über Toreingang wird Zählereingang frei gegeben.(Gated Timer). Die Funktion (Zähler/Zeitgeber), der Toreingang und der Teilerfaktor wer-

den über ein Konfigurationsregister programmiert. Die Zähler können aufwärts oder abwärts laufen, ihre Zählregister haben eine Breite von 8 bis 24 Bit. Der Maximalwert für einen Zähler mit N Bit liegt bei $Z_{max} = 2^N - 1$, das nächste Inkrement führt zum Zählerüberlauf (Abb. 8.2).

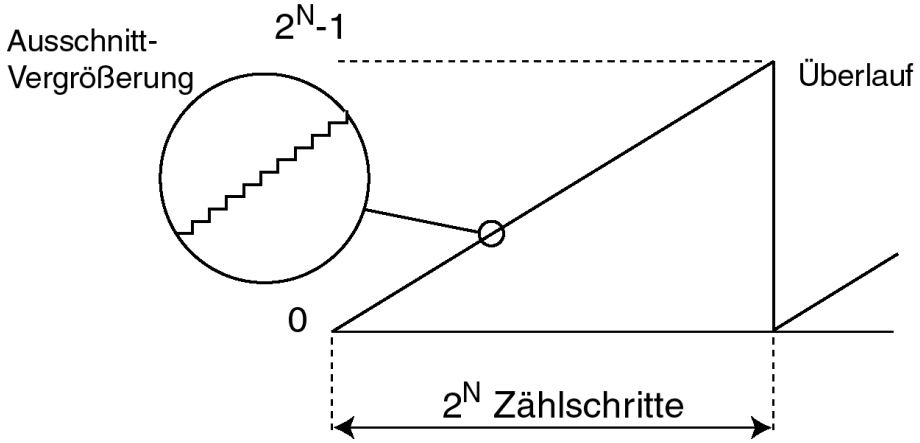


Figure 8.2: Ein aufwärts laufender Zähler beginnt nach dem Überlauf wieder bei 0

Zugriffe und Ereignisse

(Typische Architektur)

Auslesen Durch einfachen Softwarebefehl wird der aktuelle Zählerstand auf eine Variable übertragen.

Überlauf bei Z_{max} Zählerbaustein kann einen Interrupt auslösen.

Compare-Ereignis (Gleichheit des aktuellen Zählwertes mit dem Wert im Compare-Register) Zählerbaustein kann ebenfalls einen Interrupt auslösen oder einen Pegelwechsel an einer IO-Leitung bewirken. Manche Betriebsarten stellen nach dem Compare-Ereignis den Zähler auf 0.

Capture Durch ein externes Signal getriggert wird der momentane Wert des Zählregisters ausgelesen und in ein Capture-Register übertragen.

Die Interrupt-Service-Routine kann auch die Einstellungen des Timers ändern, kompliziert! Die Zeit von einem Zählerüberlauf zum nächsten ist bei einem Zählregister mit N Bit und einer Zählerinkrementzeit T_I

$$T = (2^N - Startwert) \cdot T_I$$

Typische Verwendungen eines Zählers sind:

- Zählung von Impulsen,
- Messen von Zeiten,
- Erzeugung von Impulsen,
- Erzeugung von pulsweitenmodulierten Signalen.

8.2 Anwendungsbeispiele Zählbetrieb (Counter)

In diesen Beispielen nehmen wir Folgendes an:

- 16-Bit-Zähler/Zeitgeber
- Zähler läuft aufwärts

Impulszählung, Impulsdifferenzen

Motivation: Bewegliche Teile in Maschinen, werden häufig mit so genannten Encodern ausgestattet. Diese geben bei Bewegung eine bestimmte Anzahl von Impulsen pro mm oder Winkelgrad ab. zur Erfassung diese Pulse kann man den Baustein auf Zählerbetrieb einstellen und den Zählerstand mehrfach auslesen (Abb. 8.3).

Vorgehensweise

- Timer als Zähler konfigurieren, externe Impulse durchleiten zum Zähler
- Interrupt aktivieren für den Fall des Überlaufs, Interruptservice-Routine zählt Überläufe
- Letzten Zählerstand speichern
- Neuen Zählerstand einlesen
- Differenz bilden, Anzahl Überläufe einrechnen, man erhält Gesamtzahl Impulse
- Daraus Bewegungsstrecke oder -winkel berechnen

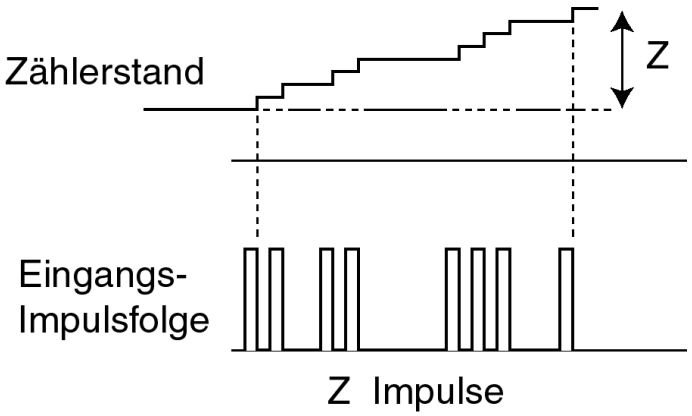


Figure 8.3: Impulszählung mit einem Zählerbaustein. Jeder einlaufende Impuls erhöht den Zählwert um eins, mit dem Capture-Befehl kann das Zählregister jederzeit ausgelesen werden.

Für eine Geschwindigkeitsmessung kann ein zweiter Timer als Zeitgeber betrieben werden und zyklisch einen Interrupt erzeugen. Man muss dann nur den ermittelten Weg durch die verstrichene Zeit dividieren, um die Geschwindigkeit zu erhalten.

8.3 Anwendungsbeispiele Zeitgeberbetrieb (Timer)

In diesen Beispielen nehmen wir Folgendes an:

- 16-Bit-Zähler/Zeitgeber
- Zählfrequenz 1 MHz
- Zähler läuft aufwärts

Übung: 8.1 Zählerüberlauf

Berechnen Sie nach welcher Zeit der beschriebene Zähler einen Überlauf hat?

Impulsabstandsmessung, Frequenzmessung

Motivation: Es gibt Sensoren, die eine Impulsfolge aussenden und ihr Messergebnis durch den Abstand der Impulse ausdrücken. Dies stellt eine sehr robuste und störungssichere Übertragung dar und befreit von allen Problemen, die mit der Übertragung analoger Messsignale verbunden sind. Gerade im industriellen Umfeld ist dies ein großer Vorteil.

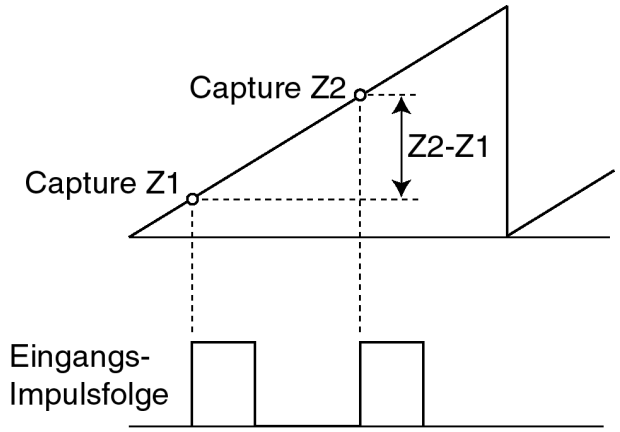


Figure 8.4: Ermittlung des zeitlichen Abstands zweier Impulse durch Einlesen der Zählwerte bei steigender Flanke

Vorgehensweise

- Impulsabstand muss deutlich größer sein als die Dauer des Zählerinkrements
- Zähler/Zeitgeber wird als Zeitgeber betreiben
- Bei der ersten ansteigenden Signalflanke wird der Zählwert ausgelesen (Capture oder Lesebefehl) oder der Zähler auf Null gesetzt.

- Bei der zweiten ansteigenden Flanke wieder Zähler auslesen (Abb. 8.4)
- Differenz der beiden Zählwerte bilden
- Differenz mit Zählerinkrementzeit malnehmen ergibt gesuchten Impulsabstand
- Evtl. Zählerüberlauf softwaremäßig behandeln.

Zahlenbeispiel Das erste Lesen ergibt einen Zählwert von 5800, das zweite 7450, es fand kein Zählerüberlauf statt. Der Impulsabstand ist $(7450-5800)\cdot 1\mu\text{s} = 1.65\text{ms}$.

Als Kehrwert des gemessenen Impulsabstandes ergibt sich sofort die Impulsfrequenz. Eine Impulslängenmessung kann ebenso erfolgen, nur muss das zweite Capture nach der fallenden Flanke des Impulses durchgeführt werden.

Zyklische Interrupts

Motivation: Um ein System oder Teile davon (z.B. die Anzeige) regelmäßig durch den Mikrocontroller zu aktualisieren, kann ein zyklischer Interrupt benutzt werden, der regelmäßig durch den Überlauf eines Zeitgebers ausgelöst wird. (=festes Zeitgerüst)

Vorgehensweise

- Zeitgeber im Compare-Modus betreiben
- Comparewert = Zykluszeit / Zählerinkrementzeit eintragen
- Interrupt für den Zähler bei Gleichheit mit Compare-Register aktivieren
- Regelmäßig wenn die Zykluszeit vergangen ist: Interrupt
- Interrupthandler würde die gewünschte Aktualisierung des Systems vor

Zahlenbeispiel Ein Compare-Wert von 50000 führt dazu, dass nach 50000 Zählschritten das Compare-Ereignis stattfindet, das bedeutet nach $50000 \cdot 1\ \mu\text{s} = 50\,ms$.

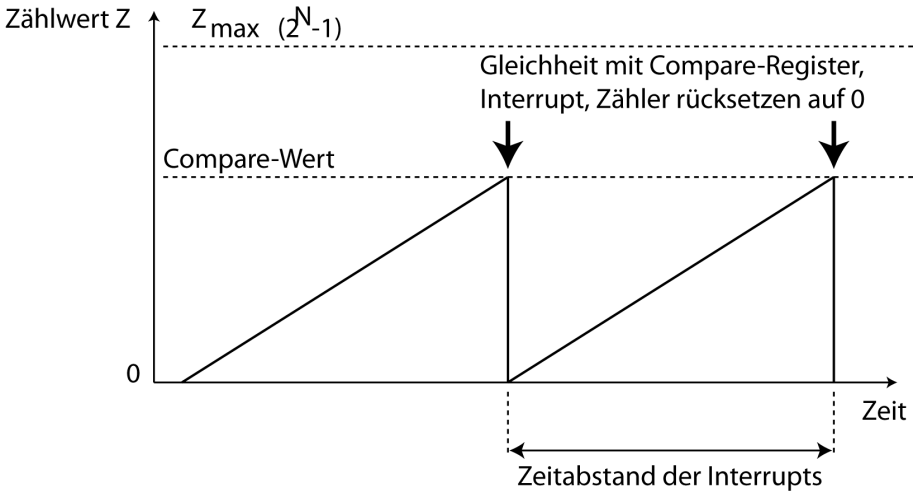


Figure 8.5: Der Überlauf eines Zeitgebers kann benutzt werden um zyklische Interrupts auszulösen.

Wenn die Zykluszeit auch mit Compare-Wert 65535 noch zu klein ist, gibt es folgende Möglichkeiten:

- Auf langsameren Zählertakt wechseln
- Vorteiler aktivieren
- In der Interrupt-Service-Routine mitzählen und erst nach mehreren Interrupts die Systemaktualisierung vornehmen.

Pulsweitenmodulation

Motivation: Mikrocontroller müssen angeschlossene Geräte. z.B. Motoren oder LEDs mit Teilleistung ansteuern. Eine Lösung mit einem Digital-Analog-Wandler und einem Verstärker ist aufwändig und teuer. Einfacher ist es, ein *pulsweitenmoduliertes Signal* (*PWM-Signal*) zu benutzen. Prinzip: Die Versorgungsspannung wird ständig für kurze Zeit abgeschaltet. Das geht so schnell, dass der Motor nicht ruckelt. (Nötigenfalls wird Tiefpassfilter zwischengeschaltet) Durch die zeitweilige Abschaltung wird weniger Leistung übertragen.

Beispiel: Einem Motor der zyklisch für 3ms eingeschaltet und danach für 1ms ausgeschaltet wird, fehlen 25 % der Leistung, der Motor läuft also mit 75 % Leistung. Allgemein ergibt sich das Tastverhältnis V zu

$$V = \frac{T_H}{T}$$

T_H Zeitspanne ist in der das PWM-Signal HIGH ist

T Zykluszeit des PWM-Signals

Vorgehensweise Es gibt mehrere Möglichkeiten, z.B.

- Man wählt eine IO-Leitung aus, die vom Zähler geschaltet werden kann
- Man schaltet die Leitung auf LOW und startet den Zähler bei 0.
- Man benutzt ein Vergleichsregister (z.B. Compare-Register0), um bei Gleichheit die Leitung mit dessen auf HIGH zu schalten
- Man benutzt ein zweites Vergleichsregister (z.B. Compare-Register 1), um bei Gleichheit die Leitung wieder auf LOW zu schalten und den Zähler auf 0 zurück zu stellen. (8.6).
- Das PWM-Signal wird selbstständig erzeugt – keine CPU-Aktivität mehr nötig!

Beispiel Der Wert in Compare-Register 1 sei 500, das ergibt eine Zykluszeit von $500\ \mu$s, also ein Signal von 2 kHz. Der Wert in Compare-Register 0 sei 200, dann wird immer $200\ \mu$s nach dem Start das Signal von LOW auf HIGH geschaltet und bleibt dann weitere $300\ \mu$s auf HIGH bis zum nächsten Zählerüberlauf mit Reload. Von insgesamt $500\ \mu$s ist das Signal also $300\ \mu$s HIGH, d.h. das Tastverhältnis ist

$$T_V = \frac{300\ \mu\text{s}}{500\ \mu\text{s}} = 0.6$$

Das PWM-Signal stellt also hier 60% Leistung zur Verfügung.

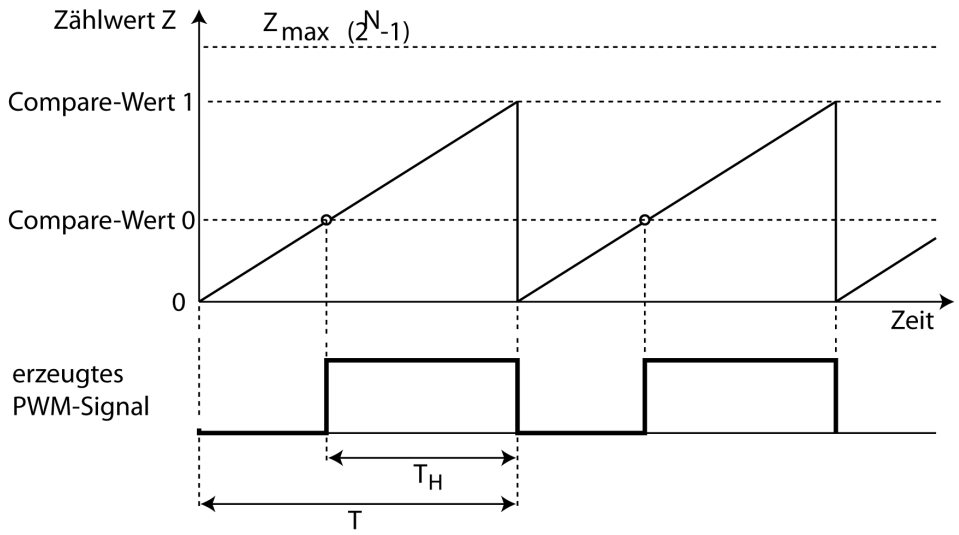


Figure 8.6: Erzeugung eines PWM-Signals: Der Compare-Wert 1 bestimmt die Zykluszeit T, der Compare-Wert 0 das Tastverhältnis V.

Fallbeispiel: Der Timer_A MSP430 von Texas Instruments

Aufbau:

- 16-Bit-Zählregister
- vier Betriebsarten
- Eingangsteiler
- Wahlmöglichkeit zwischen vier Takteingängen
- Am eigentlichen Zähler hängen mehrere komplexe Capture/Compare-Einheiten (TACCR), die den aktuellen Inhalt des Zählers nach jedem Zählschritt auswerten.
- An den Capture/Compare-Einheiten hängen Output-Units, die direkt eine Ausgangsleitung schalten können (für Signalerzeugung, z.B. PWM)
- Timer B ist ähnlich

Das Kernstück des Timers ist ein Zähler mit einem Impulseingang. Mit jedem Impuls am Eingang erhöht sich der Zählerstand um 1 oder erniedrigt sich um 1. Timer A hat ein 16-Bit-Zählregister, bei Timer B ist es per Software einstellbar. Der Zähler kann per Software

- gestartet werden
- gestoppt werden
- ausgelesen werden
- zurückgesetzt werden auf 0

Softwaremäßig kann eingestellt werden, aus welcher Quelle die Impulse am Eingang kommen sollen. Es gibt zwei Möglichkeiten:

1. Die Impulse kommen von einer externen Quelle, beispielsweise von einem bewegten Teil einer Maschine. Der Zählerstand ist dann ein Maß für eine zu

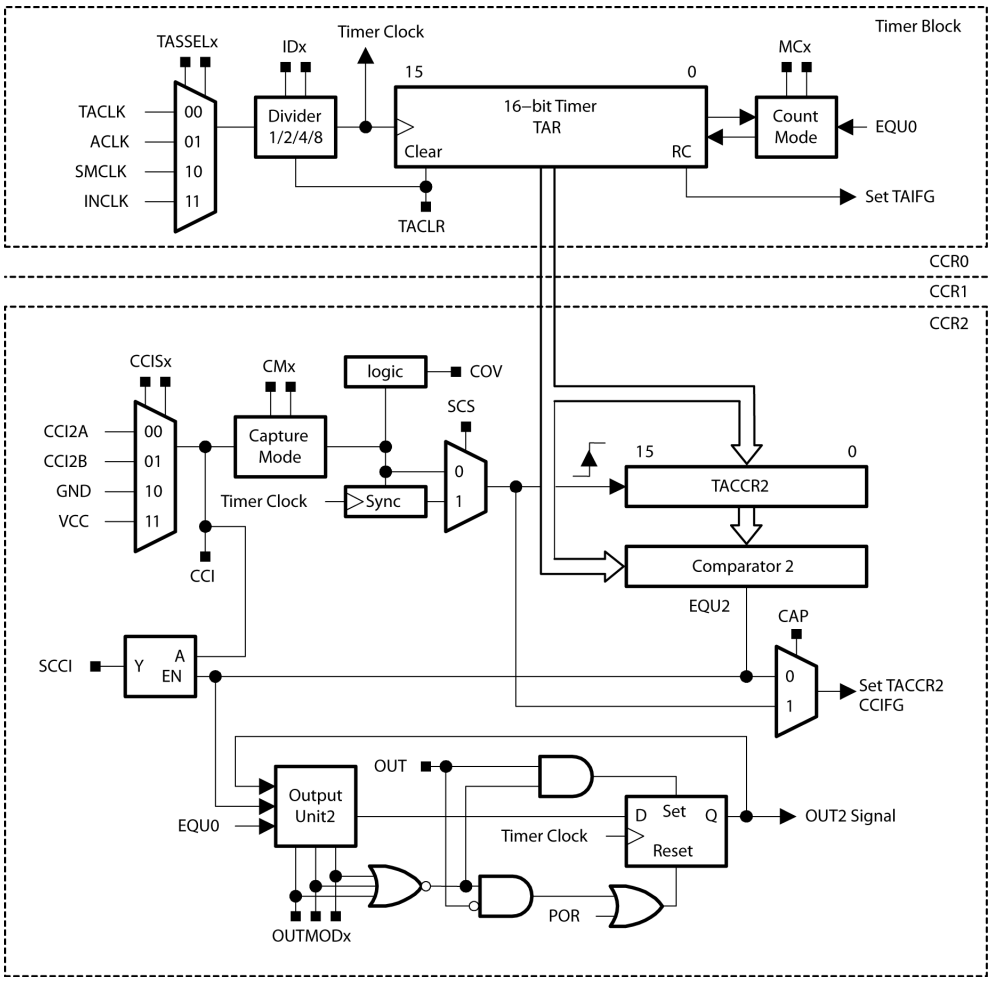


Figure 8.7: Timer_A besteht aus der eigentlichen Zählereinheit mit vier Eingängen und einem Vorteiler sowie mehreren daran angeschlossenen Capture/Compare-Einheiten, die den aktuellen Zählerstand auswerten. (Aus dem MSP430 Family User's Guide mit freundlicher Genehmigung von Texas Instruments)

messende Größe, z. B. Weg oder Winkel; der Zähler läuft im Zählerbetrieb. (Abbildung unten)

2. Die Impulse kommen von einem der internen Busse mit einer bekannten Taktfrequenz. Dann ist der Zählstand ein Maß für die Zeit, die vergangen ist; der Zähler läuft im Zeitgeberbetrieb.

Schauen wir uns die Betriebsarten der Zählereinheit an:

Halt Mode Zähler steht

Continuous Mode Zähler zählt aufwärts bis zum Endwert 0xFFFF; beim nächsten Takt springt er auf 0000 und zählt dann wieder aufwärts bis zum Endwert.

Up Mode Zähler zählt aufwärts bis zu dem Wert, der im TACCR0 (Timer A Capture/Compare-Register 0) hinterlegt ist; beim nächsten Takt springt er auf 0000 und zählt dann wieder aufwärts bis zum Endwert in TACCR0.

Up/Down Mode Zähler zählt aufwärts bis zu dem Wert, der im TACCR0 (Timer A Capture/Compare-Register 0) hinterlegt ist; ab dem nächsten Takt zählt er abwärts bis 0000, dann wieder aufwärts u.s.w.

Für die Programmierung des Timer_A ist vor allem das Steuerregister TACTL (Timer A Control) entscheidend. Darin liegen folgende Bitfelder:

TASSEL Timer A Source Select: Zugeführter Takt ist TAClock, AClock, SMClock oder InClock

ID Input Divider: Teiler 1, 2, 4 oder 8

MC Mode Control: Halt, Continuous, Up oder Up/Down

TACLR Timer A Clear, setzt den Timer auf 0000, wenn mit 1 beschrieben

TAIE Timer A Interrupt Enable, Freischaltung des Interrupts durch Timer

TAIFG Timer A Interrupt Flag, zeigt an, ob ein Interrupt ausgelöst wurde

Übung: 8.2 Timer A konfigurieren

Auf das Control Register des Timer A wird mit der Wert 0x0256 geschrieben. Wie arbeitet danach der Timer?

Einfachste Nutzung des Zählers: Einfach auslesen, auch wenn er läuft: (`Variable = TAR`)

Nutzung mit Interrupt:

- Auslösung eines Interrupts durch , wenn Wert im TACCR0 gleich Zählwert. (Compare-Ereignis)
- Regelmäßige Interrupts möglich, Zeitgerüst

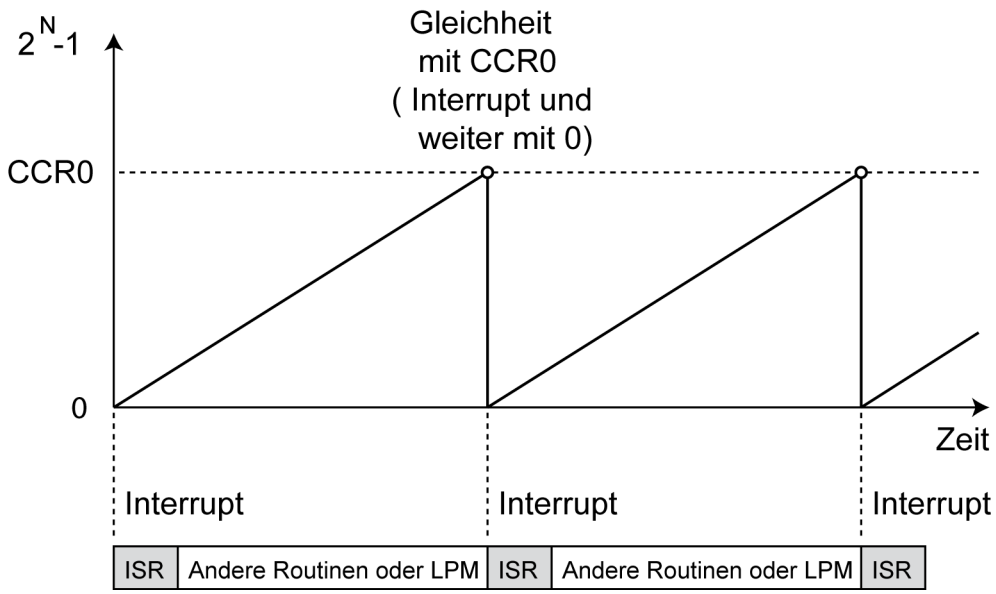


Figure 8.8: Mit dem Zähler des MSP430 im Up Mode lässt sich einfach ein Zeitgerüst für zyklisch ausgeführte Programmteile aufbauen

Programmbeispiel:
Blink-Programm, durch den Timer_A gesteuert, Takt=AClock (32768 Hz). Vergleichsregister TACCR0 = 32768, nach exakt einer Sekunde die Gleichheit erreicht. Zähler läuft im Up Mode, beginnt nach Interrupt wieder bei, daher zyklisch nach einer Sekunde ein Interrupt.

```
/* ************************************************************
  Beispielprogramm blink2.c
  Lässt auf dem Board eine LED endlos blinken
  Zeitraster mit Hilfe des Interrupts von Timer_A

  Kommentar zur Schaltung auf Board:
  Leuchtdioden an P1.0 - P1.7 leuchten wenn Ausgang=L (0)

*/
#include  <msp430x22x2.h>

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD; // Stop  Watchdog Timer

  TACTL = TASSEL_1 + TACLR; // Beschreiben des TimerA-Controlregisters:
                        // - TimerA Source Select = 1 (Eingangstakt ist AClock)
                        // - Clear TimerA-Register  (Zählregister auf 0 setzen)
                        // Input Divider=1, Timer ist im Halt Mode

  TACCTL0 = CCIE;       // Capture/Compare-Unit0 Control-Register beschreiben:
                        // Interrupt-Auslösung durch Capture/Compare-Unit0
                        // freischalten (CCR0)
  TACCR0 = 32768;       // Capture/Compare-Register 0 mit Zählwert beschreiben
```

```
  P1SEL |= 0x00;        // P1 hat Standard-IO-Funktion
  P1DIR |= 0x01;        // P1.0 ist Ausgang

  TACTL |= MC_1;             // Start Timer_A im up mode (Mode Control = 1)
  __enable_interrupt();      // enable general interrupt
                             //   (Interrupts global freischalten)

  __low_power_mode_0();    // low power mode 0:

  while (1);
}

// Timer A0 interrupt service routine
// wird jedesmal aufgerufen, wenn Interrupt CCR0 von TimerA kommt
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A0 (void)
{
  P1OUT ^= 0x01;           // Leitung 0 von Port 1 toggeln
}
```

Die Funktionen des Timers bleiben auch im Low-Power-Mode erhalten, wenn nicht der Takt abgeschaltet wird. Das nutzt man in der Regel zu einem stromsparenden Betrieb aus. Man schaltet den Mikrocontroller in den Low-Power-Mode und lässt ihn durch einen Tasten- oder Timerinterrupt aufwecken. Aber Achtung: Nicht den Takt abschalten, der den Timer versorgt.

Capture

Capture wird freigeschaltet, eine Eingangsleitung bestimmt. Wenn externe Ereignis kommt (steigende oder fallenden Flanke) wird der Inhalt des Zählregisters direkt in das TACCR0 kopiert und das Flag TAIFG gesetzt. Keine CPU-Aktivität, kein Interrupt.

Beim MSP430 sind diese Zusatzfunktionen zu einer Capture/Compare-Einheit mit integrierter Output-Unit zusammengefasst. Da man heute großzügig denkt, besitzt der Timer A 3 Capture/Compare-Einheiten, die aber alle an ein einziges Zählregister angekoppelt sind. Der Timer B besitzt sogar 7 Capture/Compare-Einheiten.

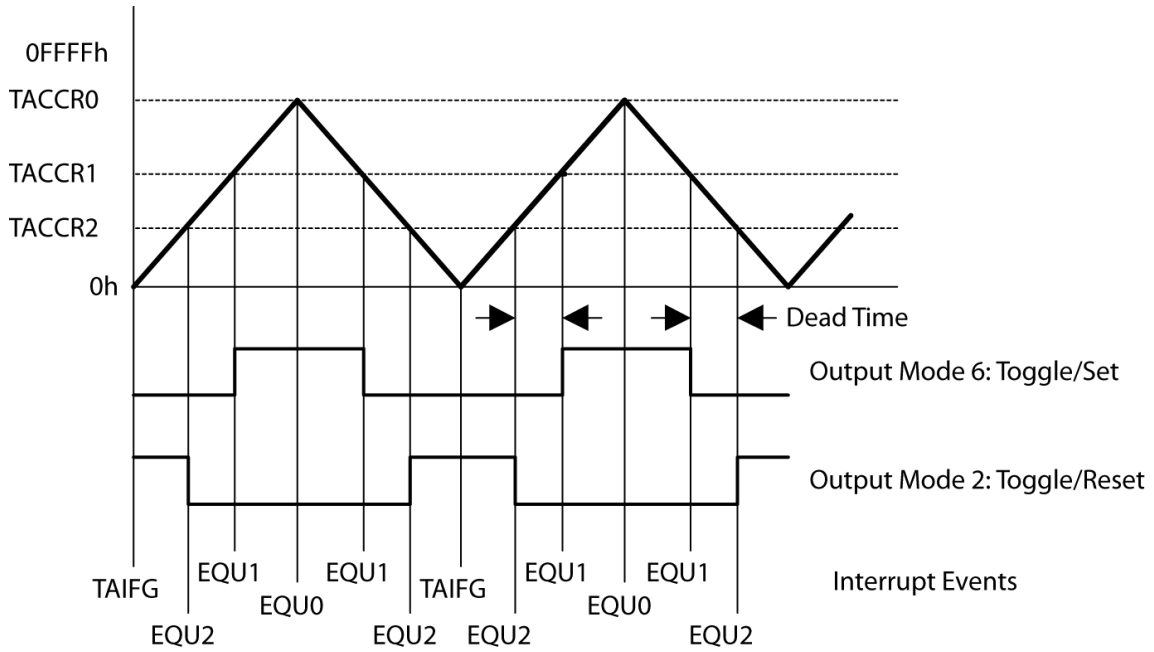


Figure 8.9: Die Output-Modi der Capture/Compare-Einheiten können benutzt werden um komplexe Ausgangssignale ohne CPU-Aktivität zu erzeugen. Hier läuft der Zähler im Up/Down-Mode. (Aus dem MSP430 Family User's Guide mit freundlicher Genehmigung von Texas Instruments)

Übung: 8.3 Ausgabe Analogsignal

Wie kann man mit einem Mikrocontroller ohne Digital/Analog-Wandler ein Analogsignal ausgeben

Übung: 8.4 Peripherie-Auswahl

In einem Wäschetrockner, der mit dem Mikrocontroller MSP430 gesteuert wird, soll der Feuchtefühler einmal pro Sekunde ausgelesen werden. Geben Sie an, welche Baugruppen des Mikrocontrollers man verwenden sollte und welche Programmierungsschritte notwendig sind.

PRAKTIKUMSAUFGABEN NACH DIESEM ABSCHNITT

1. *Aufgabe 7 Zyklische Interrupts durch den Timer*
2. *Aufgabe 8 Abwärts laufende Uhr mit akustischem Alarm (Tea-Timer)*
3. *Aufgabe 9 Taschenrechner mit Interrupts*
4. *Aufgabe 10 Impulse von einer rotierenden Welle zählen und anzeigen*
5. *Aufgabe 11 Anzeige der Drehzahl des rotierenden Rades*
6. *Aufgabe 12 Motoransteuerung*

9 Mikrocontroller: Verarbeitung analoger Signale

9.1 Analoge Signale

Analoge Größen haben einen kontinuierlichen, lückenlosen Wertebereich. Innerhalb des Wertebereiches ist also jeder beliebige Zwischenwert möglich. Beispiele:

- Temperatur
- Geschwindigkeit
- Masse, Gewicht
- Strom, Spannung
- Helligkeit

Analoge Größen werden oft linear in einander umgewandelt, Beispiel:

Temperatursensor $0^oC \ldots 100^oC \rightarrow 0V \ldots 5V$

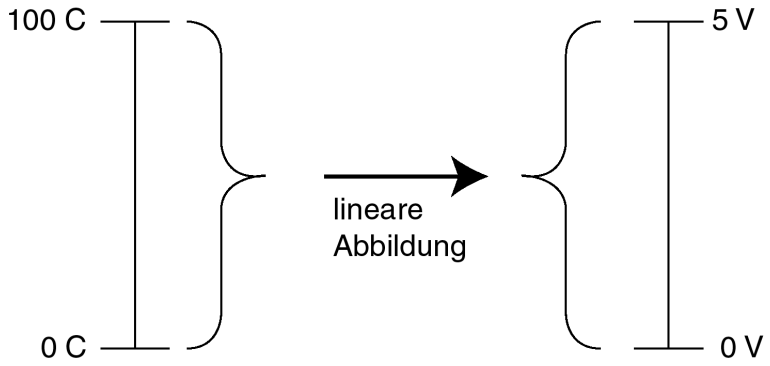


Figure 9.1: Ein Sensor bildet einen analogen Wertebereich von Temperaturen auf einen analogen Wertebereich von Spannung ab.

Bei diesem Sensor entspricht zum Beispiel 40^oC einer Ausgangsspannung von $2V$ und 31.7^oC entspricht $1.585V$.

Übung: 9.1 Analoge Größen

1. Nennen Sie zwei weitere analoge Größen!
2. Welche Temperatur liegt vor, wenn obiger Sensor 3.2V abgibt?

- In Embedded Sytems ist es oft erforderlich, analoge Signale zu verarbeiten, z.B. von Messfühlern (Sensoren).
- Analoge Signale können nicht über IO-Ports ein- und ausgegeben werden, diese verarbeiten ja digitale Spannungspegel.
- Dazu haben die meisten Mikrocontrollern einen *Analogeingang*.
- Mikrocontroller mit einem *Analogausgang* können analoge Signale erzeugen; eine Alternative dazu sind PWM-Signale.

9.2 Analog-Digital-Umsetzer

Analog-Digital-Umsetzer, abgekürzt ADU, (engl. Analog/Digital-Converter, ADC)

- haben einen analogen Eingang
- haben N digitale Ausgangsleitungen
- Man spricht auch von einer Auflösung von N Bit.
- verwandeln ein analoges Signal in einen digitalen Wert: Ein Bitmuster auf N digitalen Ausgangsleitungen (Abb. 9.2)
- Die Ausgangswerte liegen bei N Bit Auflösung im Bereich $0 \ldots 2^N - 1$.

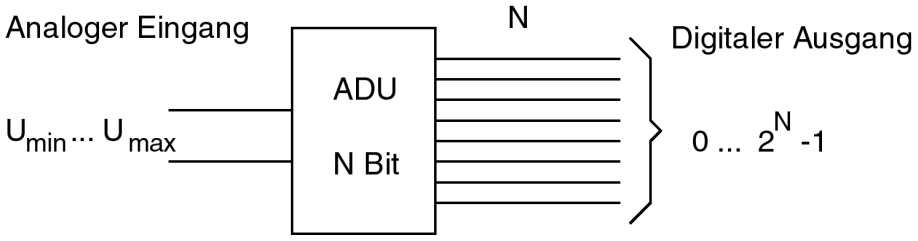


Figure 9.2: Ein Analog-Digital-Umsetzer

Die Umsetzung ist eine lineare Abbildung nach der Formel:

$$Z = \frac{U_e - U_{min}}{U_{max} - U_{min}}(2^N - 1)$$

Die Höhe der Quantisierungsstufen U_{LSB} hängt von der Auflösung des Analog-Digital-Wandlers ab:

$$U_{LSB} = \frac{U_{max} - U_{min}}{2^N - 1}$$

Übung: 9.2 Analog/Digital-Wandlung

An einem ADU hat 10 Bit Auflösung und einen Eingangsbereich von $0..5V$. Welcher Digitalwert wird ausgegeben, wenn die Eingangsspannung 0V, 5V, 1V oder 3.5V ist?
Wie hoch sind die Quantisierungsstufen?

Eigenschaften von Analog-Digital-Umsetzern:

- Jeder ADU braucht eine gewisse Zeit für den Wandlungsvorgang. Diese Wandlungszeit begrenzt auch die maximal mögliche Anzahl Wandlungen pro Sekunde, die *Abtastfrequenz*.
- Die Wandlungszeit hängt stark von der Bauart des Wandlers ab.
- Alle ADU brauchen eine externe Referenzspannung und bestimmen das Wandlungsergebnis durch Vergleich mit dieser Referenzspannung.

Alle Analog-Digital-Umsetzer sind mit Fehlern behaftet.

- Der *Quantisierungsfehler* entsteht durch die Rundung auf eine ganze Zahl. Der Quantisierungsfehler beträgt auch bei einem idealen ADU bis zu $0.5U_{LSB}$.
- Der Linearitätsfehler entsteht durch Fertigungstoleranzen.
- Statistische Schwankungen entstehen durch Rauschen

Analog-Digital-Umsetzer bei Mikrocontrollern

- Viele Mikrocontroller haben einen ADU On-Chip, in der Regel mit einer Auflösung zwischen 8 und 12 Bit.
- Diese ADUs wandeln meist nach dem Verfahren der sukzessiven Approximation.
- Manchmal lässt sich U_{min} und U_{max} in Schritten programmieren. Damit kann der Eingangsbereich aufgespreizt und an das analoge Signal angepasst werden.
- Das ebenfalls erforderliche Abtast-Halte-Glied ist meist auch On-Chip; es sorgt dafür, dass das Signal für den Zeitraum der Wandlung zwischengespeichert wird und konstant bleibt.

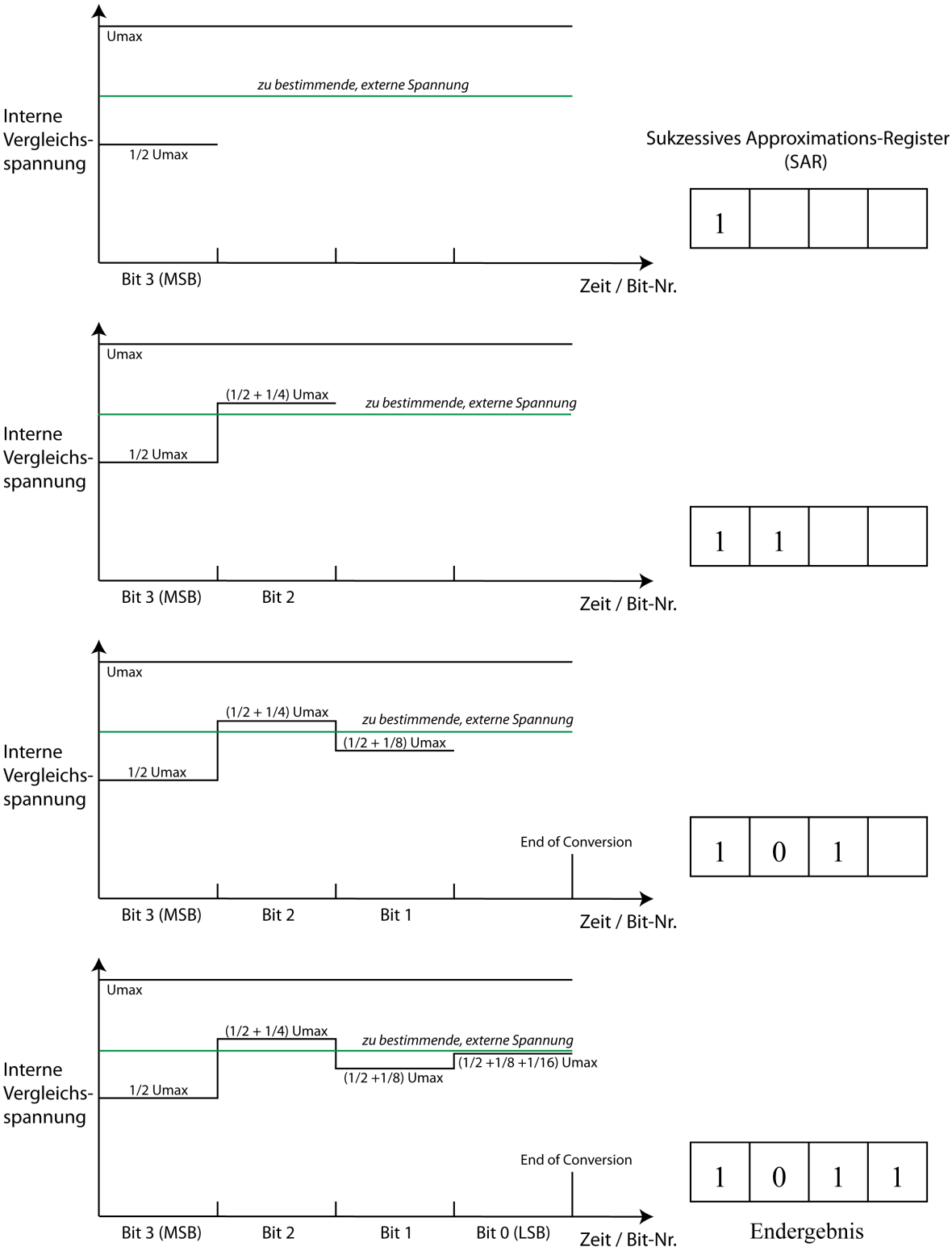


Figure 9.3: Das Wandlungsverfahren der sukzessiven Approximation am Beispiel eines 4-Bit-AD-Wandlers. (Ausgabebereich: 0–15, $U_{LSB} = \frac{1}{16} U_{max}$)

9.3 Digital-Analog-Umsetzer

Digital-Analog-Umsetzer, abgekürzt DAU, (engl. Digital/Analog-Converter, DAC) arbeiten genau umgekehrt wie ADUs: Sie erhalten an ihren digitalen Eingangsleitungen eine Ganzzahl in binärer Darstellung und erzeugen am Ausgang die entsprechende analoge Spannung. Diese ergibt sich gemäß:

$$U_a = \frac{Z}{2^N - 1}(U_{max} - U_{min}) + U_{min}$$

Es sind nur wenige Mikrocontroller mit On-Chip-DAU ausgerüstet, es bleiben aber die Alternativen eines externen DAU oder des PWM-Verfahrens.

Fallbeispiel: Der 10-Bit-Analog/Digital-Wandler ADC10 des MSP430

Der Aufbau des 10-Bit-ADC ist in Abb. 9.4 vereinfacht gezeigt. Die analogen Signale kommen von außen über einen oder mehrere der analogen Kanäle A0 – A7 oder A12 – A15 (nicht auf allen MSP430 verfügbar) herein. Der Eingangsmultiplexer wählt einen Kanal aus, den er weiter leitet auf den *Sample-and-Hold-Baustein* (S&H).

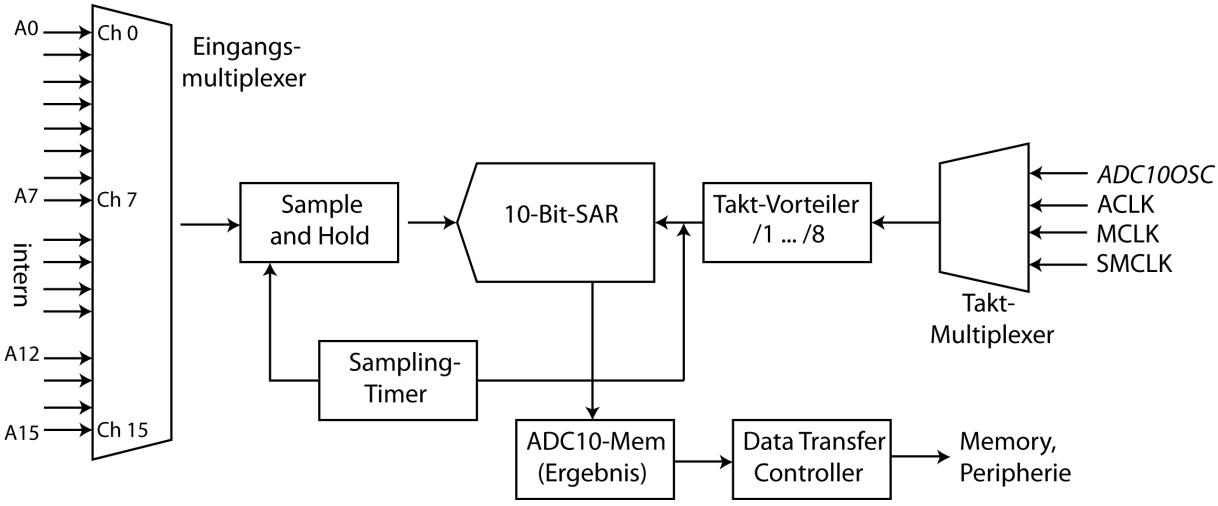


Figure 9.4: Vereinfachte Darstellung des 10-Bit-ADC im MSP430.

Einen DAC besitzt unser MSP430F2272 nicht, wenn man ein analoges Signal ausgeben will, muss man die PWM-Methode benutzen. Er ist aber mit einem ADC ausgestattet, der nach dem Verfahren der Sukzessiven Approximation arbeitet. Bei diesem Verfahren erhält der ADC ein Taktsignal und ermittelt das Ergebnis in mehreren Schritten, die durch diesen Takt getriggert werden.

Im MSP430F2272 besteht die Kette zum ADC aus den Bestandteilen Multiplexer, Tiefpass, Abtast-Halt-glied (Sample and Hold). Mit dem Multiplexer kann einer von 16 Eingangskanälen ausgewählt werden. Es sind allerdings zwei Kanäle unbenutzt und zwei werden intern benutzt, somit bleiben nur 12 nach außen geführte analoge Eingänge. Der Tiefpass ist zur Unterdrückung von Alias-Effekten eingefügt. Das Abtast-Halte-Glied erfasst über eine gewisse Zeit das Eingangssignal (Sample) und

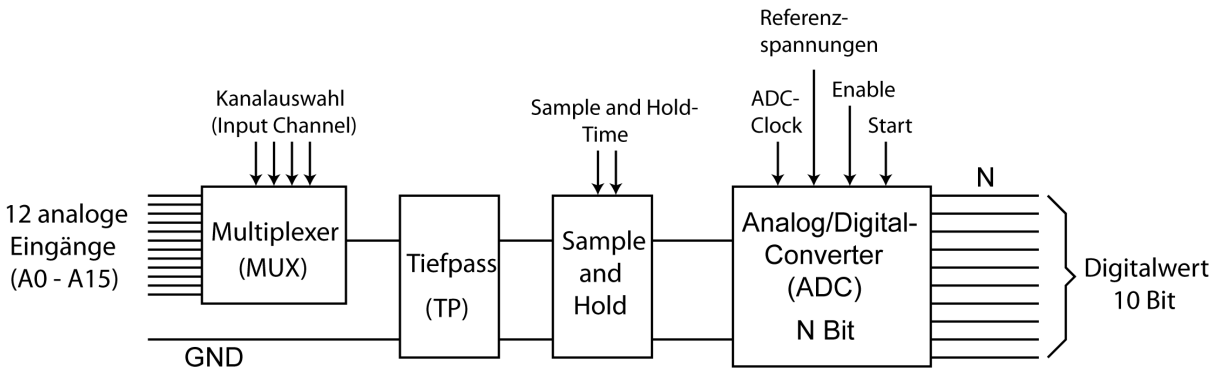


Figure 9.5: Aufbau der ADC-Signalkette im MSP430.

hält es dann konstant fest (Hold), bis der ADC mit der Wandlung fertig ist. Die Erfassungszeit des Abtast-Halte-Gliedes kann eingestellt werden.

Der ADC des MSP430F2272 hat eine Auflösung von 10 Bit und einen variablen Eingangspannungsbereich. Die Wandlungszeit hängt davon ab, mit welchem Takt $f_{ADC10Clock}$ man den ADC versorgt. Benutzt man den eingebauten ADC-Taktgeber, so beträgt die Wandlungszeit $2 \ldots 3.5\,\mu s$. Benutzt man eines der anderen Taktsignale, so steht auch wieder ein Vorteiler zur Verfügung. Die Wandlungszeit ist dann:

$$t_{ADC} = 13 * ADCDIV * 1/f_{ADC10Clock} + t_{SH}$$

In jedem Fall muss man nach dem Start (Start of Conversion) der Wandlung abwarten, bis das Ergebnis zur Verfügung steht. Ablauf einer Wandlung:

- Die Wandlung wird durch das Steuerbit ADC10SC (Start of conversion) ausgelöst
- Wandler ist jetzt beschäftigt, was durch das Flag ADC10Busy angezeigt wird.
- Der Sample-and-Hold-Baustein sampelt (registriert und mittelt) das Signal über einen bestimmten Zeitraum, die Sampling-Zeit. Je größer die Sampling-Zeit ist, um so besser werden Signal-Störungen aller Art ausgemittelt.
- Die Sampling-Zeit kann beim ADC10 über ein Steuerregister auf 4, 8, 16 oder 64 ADC-Takte eingestellt werden.
- Nach dem Sampling wird die Verbindung zwischen Multiplexer und S&H geschlossen.
- Haltephase (Hold), der Sample-and-Hold-Baustein hält das gemittelte Signal konstant und ruhig an seinem Ausgang bereit, damit der AD-Wandler arbeiten kann
- Der Wandler braucht für jedes Bit einen Takt, so lange muss das Signal konstant bleiben.
- In das SAR-Register wird nun mit jedem Takt ein Bit eingeschrieben, wobei die Vergleichsspannung Vref benutzt wird um das Bit zu bestimmen.
- Das Eingrenzungsverfahren dabei heißt *sukzessive Approximation* (schrittweise Annäherung) und ähnelt der binären Suche
- Gesamtdauer der Wandlung: 13 Takten (Wandlung) + ein Takt für Start +

programmierte Anzahl Takte Samplingzeit.

- Die absolute Wandlungszeit hängt natürlich noch vom ausgewählten ADC10-Takt ab.
- Wenn alle 10 Bit ermittelt sind, ist die Wandlung beendet, das Flag ADC10Busy wird gelöscht und ADC10IFG wird gesetzt.
- Wenn der ADC entsprechend konfiguriert ist, wird ein Interrupt ausgelöst
- Im Ergebnisregister ADC10MEM steht Ergebnis bereit

Wichtig für die Konfiguration:

- Man darf nicht vergessen, die Referenzspannung und das ADC10-SAR einzuschalten (REFON, ADC10ON); um Strom zu sparen sind diese defaultmäßig ausgeschaltet.
- Während einer Wandlung, darf nichts verstellt werden!
- Deshalb gibt es in den beiden Steuerregistern (ADC10CTL0 und ADC10CTL1) viele Bits (in Doku grau markiert)), die nur beschrieben werden können, wenn die Wandlung noch nicht frei geschaltet ist (ENC=0)

Im folgenden Codeabschnitt ist eine einfache AD-Wandlung samt Konfiguration des ADC10 gezeigt.

```
/* ***************************************************************************

  // Codeabschnitt einfache AD-Wandlung

  // ADC10 konfigurieren
  ADC10CTL1 = INCH_6;          // Analogkanal 6 wird ausgewählt,
                               // die alternative Belegung von Digitalport P3.6
  ADC10AE0 |= BIT6;            // Analogeingang Kanal 6 freischalten

                               // ADC10 Controlregister 0 programmieren

  ADC10CTL0 = SREF_0+ADC10SHT_1+REFON+ADC10ON;
  //  ADC einstellen: SREF:        Eingangsbereich Vss .. Vcc
  //                  ADC10SHT:    Sampling-Zeit ist 8 ADC-Takte
  //                  REFON:       Referenzspannung einschalten
  //                  ADC10ON:     ADC10 einschalten
  ADC10CTL0 |= ENC;             // Enable Conversion

  ADC10CTL0 |= ADC10SC;            // ADC10 Start of Conversion
    // Wandlung läuft ...
  while((ADC10CTL0&ADC10IFG)==0);  // Warten auf Flag ADC10IFG, wenn gesetzt ist
                                   // Wandlung fertig, ADC10MEM geladen

  Ergebnis = ADC10MEM;             // Ergebnis auslesen
  //  ... Ergebnis verwerten
```

Der gerade geschilderte Vorgang ist eine einzelne Wandlung eines Kanals. Insgesamt bietet der ADC10 aber vier Möglichkeiten:

- Einfache Einzelkanal-Wandlung

- Einfache Wandlung einer Gruppe von Kanälen
- Endlos wiederholte Wandlung eines einzelnen Kanals
- Endlos wiederholte Wandlung einer Gruppe von Kanälen

Übung: 9.3 Genauigkeit Digitalisierung

Ein analoges Signal im Spannungsbereich 0..5V soll mit einer Genauigkeit von 0.3% digitalisiert werden. Kann der ADC10 des MSP430 verwendet werden?

MEHR INFORMATIONEN

→ MSP430 Family's User Guide, Abschnitt ADC10

→ Buch: Mikrocontrollertechnik

→ Buch: Das große MSP430-Praxisbuch

PRAKTIKUMSAUFGABEN NACH DIESEM ABSCHNITT

1. *Aufgabe 13 Analoges Signal vom Potentiometer ermitteln*
2. *Aufgabe 14 Park Distance Control*
3. *Aufgabe 15 ADC-Gerätetreiber*

10 Software-Entwicklung für Mikroprozessoren (Teil IV)

10.1 Programmtest

Debugging

Ein "'Bug"' ist ein Ausdruck für einen Programmfehler und fast jedes Programm enthält anfangs noch einige Bugs. (Bug=Käfer, Debug=Entwanzen) Das Debuggen ist das Aufspüren und entfernen dieser manchmal sehr versteckten Programmfehler. Wie erhält man Informationen aus dem Programm? (Eingebettete Systeme enthalten oft kein grafisches Display!)

- Über Leuchtdioden (sehr mühsam)
- Über das eingebaute Display
- Über eine serielle Leitung ausgeben und in einem Fenster auf dem Entwicklungsrechner zeigen. (geht ganz gut)
- Über einen Debugger, der sich über eine Schnittstelle mit dem Zielsystem synchronisiert; beste Lösung, geht aber nur, wenn Zielsystem dedizierte Debugschnittstelle hat – heute meistens JTAG-Schnittstelle.

In-System-Debugging. Ein guter Debugger bietet ein Debuggen im Zielsystem via JTAG-Port mit vielen Möglichkeiten:

Haltepunkte (Breakpoints) stoppen das Programm, wenn eine festgelegte Instruktion erreicht wird. Über das Debug-Interface kann dann der Inhalt von Registern und Speicher inspiziert werden, man kann also den Inhalt von Programmvariablen kontrollieren.

Inspektion Wenn System angehalten ist: Inspektion aller Speicherplätze sowie Register und Peripheriebereiche

Stepping (Einzelschrittbetrieb) führt immer nur die nächste Instruktion aus und stoppt dann wieder; so kann man den Ablauf in jedem Teilschritt verfolgen.

Ein Watchpoint (Beobachtungspunkt) stoppt das Programm, wenn eine bestimmte Variable (Speicherplatz) verändert wurde.

Manchmal komplexere Haltebedingungen möglich (Ausdrücke mit Variablen)

Ein Tracing ("'Spurverfolgung"') gibt Aufschluss über den Ablauf und die genommenen Verzweigungen im Programm

Patching Ist ein direkter Eingriff mit überschreiben von Speicherinhalt. Patching kann erfolgen, wenn das Programm gestoppt ist.

Ein Profiling ist eher ein statistisches Verfahren, es stellt die Häufigkeit der durchlaufenen Programmzweige dar.

- Breakpoints (Haltepunkte)
- Watches (Variablen werden mit aktuellem Wert angezeigt)
- Stepping Einzelschritte, Funktion in einem Schritt ausführen, Rückkehr zu aufrufenden Ebene usw.
- Peripherie-register sehen / ändern
- Sichtbarkeit des Assemblercodes

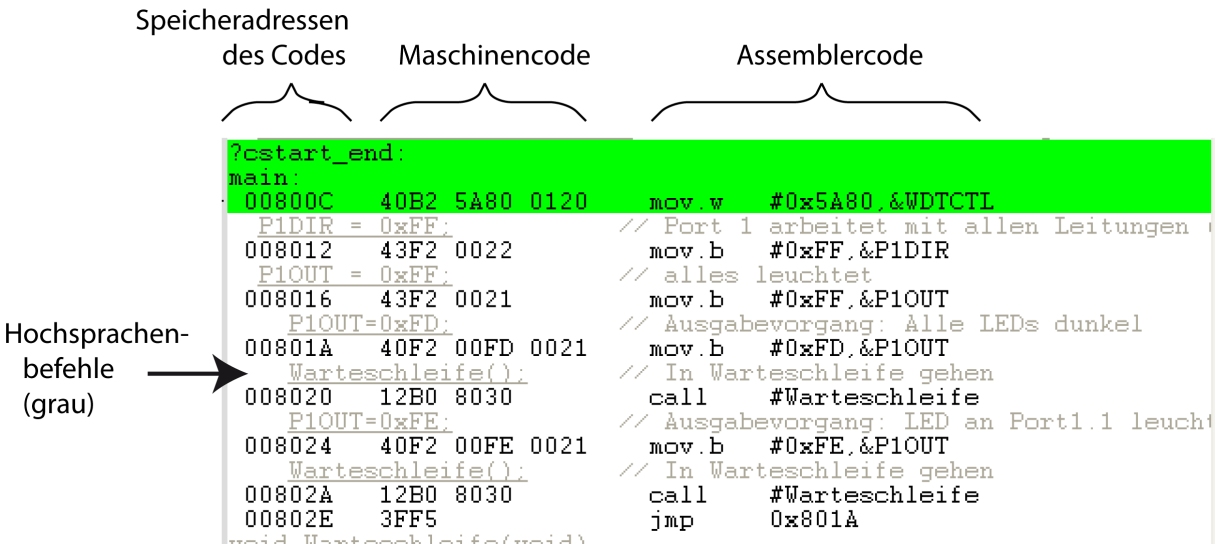


Figure 10.1: Die IDE von IAR Systems. es ist die Disassembly-View des Debug-Fensters geöffnet, man sieht den aktuellen Assembler- und Maschinencode sowie als Kommentar die Hochsprachenbefehle (C-Befehle) des Quelltextes.

Typische Vorgehensweise

- Am Beginn eines kritischen Abschnitts einen Breakpoint zu setzen
- System inspizieren
- Weiter unten eine neuen Breakpoint setzen oder im Einzelschritt weiter gehen
- Hat man entdeckt, dass eine Variable einen nicht erwarteten Dateninhalt hat, kann man den richtigen Wert in diese Variable patchen und versuchen, ob das Programm nun wunschgemäß weiter läuft.
- Nützlich: Breakpoints in Interrupt Service Routinen, um zu sehen, ob der Interrupt auslöst

Simulation

- Der *Simulator* ist ein sehr preiswertes Hilfsmittel (läuft auf PC)
- Kein echter Hardwarezugriff möglich
- Keine Echtzeit

- Eng begrenzte Möglichkeiten

In-Circuit-Emulatoren

In-Circuit-Emulatoren, kurz ICE, sind die teuersten und leistungsfähigsten Entwicklungswerkzeuge für Microcontroller.

- Ein In-Circuit-Emulator hilft auch bei schwer zu findenden Fehlern, die z.B. nur gelegentlich auftreten.
- Der ICE emuliert den Mikrocontroller vollständig, alle Signalen und echtes Zeitverhalten.
- Der echte Mikrocontroller aus dem Sockel des Zielsystems gezogen und stattdessen der Anschlussstecker des ICE eingesteckt
- Test mit fertigem Zielsystem, einschließlich aller Komponenten auf Platine

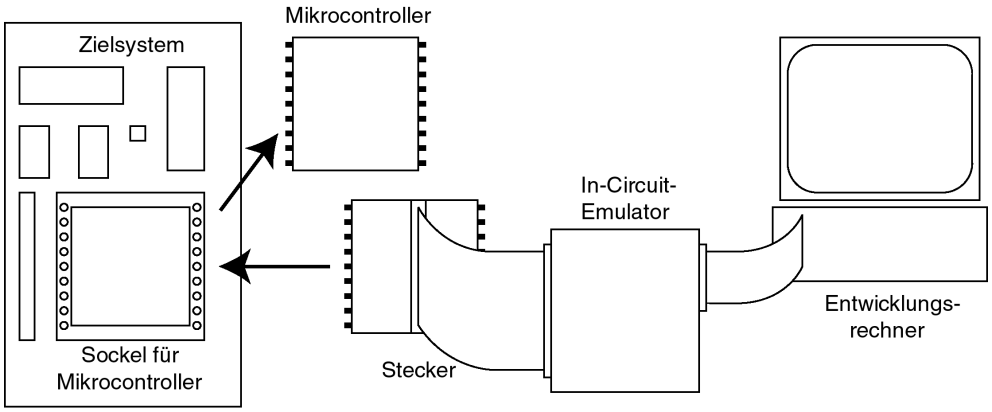


Figure 10.2: Ein In-Circuit-Emulator ersetzt in der Testphase den kompletten Mikrocontroller und bietet weitgehende Debug-Möglichkeiten.

11 Speicherbausteine

11.1 Allgemeine Eigenschaften

Speicherbausteine stellen sozusagen das Gedächtnis eines Computers dar. Manche Informationen müssen für Jahre gespeichert bleiben, wie z. B. die Laderoutinen im BIOS eines PC, andere nur für Millionstel Sekunden, wie die Schleifenvariable eines Anwendungsprogrammes. Eine Übersicht über die Halbleiterspeicher ist in Abb. 11.1 gegeben.

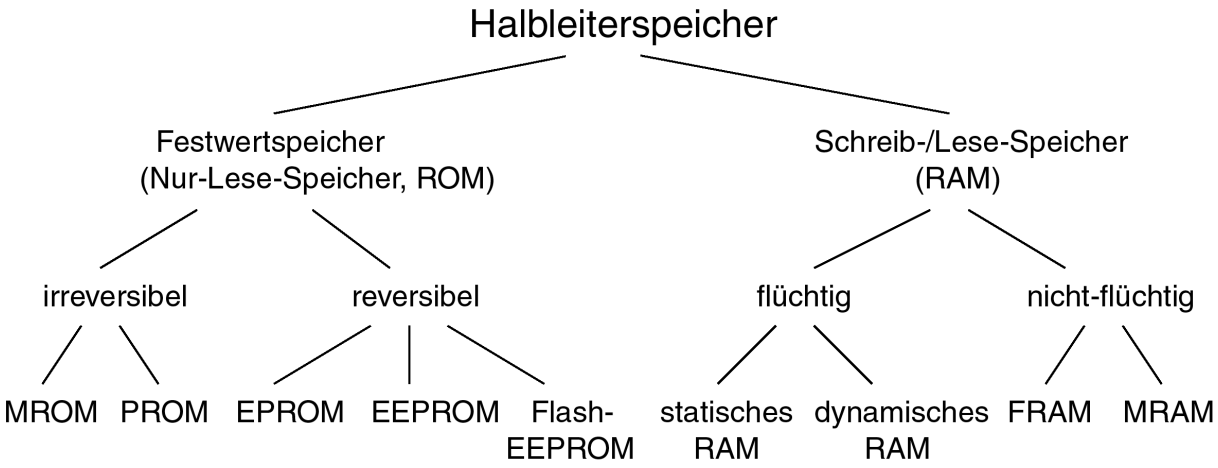


Figure 11.1: Die wichtigsten Typen von Halbleiterspeichern.

Gemeinsamkeiten der Halbleiterspeicherbausteine:

- Gitterartiger (matrixartiger) Aufbau mit waagerechten und senkrechten Leitungen
- An den kreuzungspunkten der Leitungen sitzen die Speicherzellen
- Die waagerechten Leitungen heißen *Wortleitungen*, die senkrechten *Bitleitungen*
- Jede Speicherzelle speichert 1 Bit.
- Speicherbaustein verfügen über einen Eingang zur Bausteinaktivierung, der z. B. "'Chip Select"' ($\overline{CS}$) heißt.

Wenn N die Gesamtzahl der Adressbits ist, beträgt die Anzahl der Adressen auf dem Speicherchip

$$Z = 2^N$$

Zum Auslesen einer Speicherzelle wird die am Baustein angelegte Speicheradresse geteilt (Abb. 11.3). Der eine Teil geht an den Zeilenadressdekoder, welcher dieses

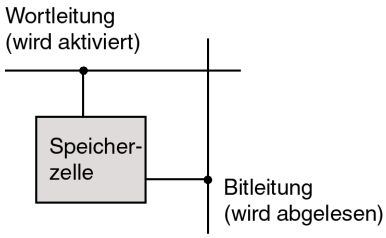


Figure 11.2:
Speicherzelle mit Wort– und Bitleitung.

Bitmuster als Zeilenadresse interpretiert, eine Wortleitung auswählt und aktiviert.

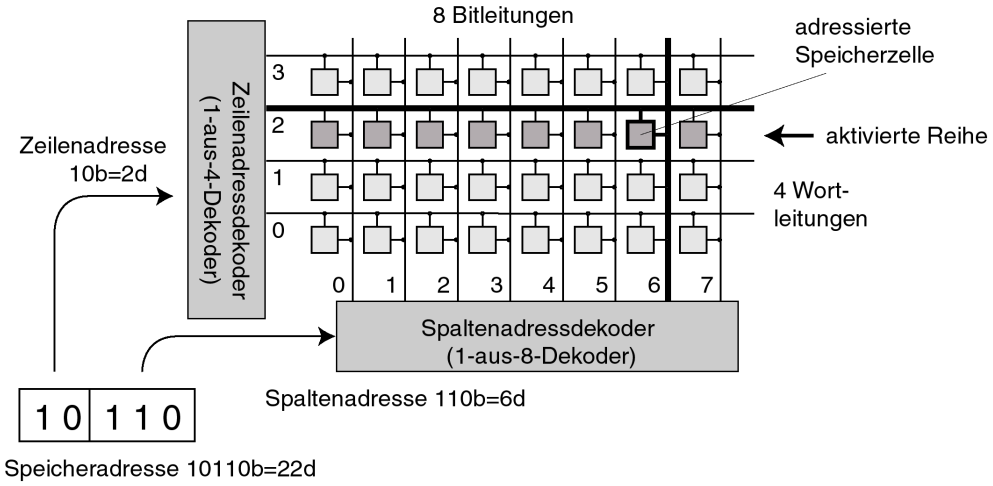


Figure 11.3: Selektieren der Speicherzelle mit der Adresse 22 in einem 32-Bit-Speicherbaustein.

Lesevorgang:

- Eine Wortleitung wird aktiviert
- Dadurch werden alle Speicherzellen aktiviert, die an diese Wortleitung angeschlossen sind
- Die aktivierten Zellen geben ihren Dateninhalt (HIGH/LOW) auf die Bitleitungen aus.
- Der Spaltenadressdekoder wählt eine der Bitleitungen aus
- Das Signal wird verstärkt und auf den Datenbus ausgegeben.

Schreibvorgang:

- Eine Wortleitung wird aktiviert
- Dadurch werden alle Speicherzellen aktiviert, die an diese Wortleitung angeschlossen sind
- Die aktivierten Zellen geben ihren Dateninhalt (HIGH/LOW) auf die Bitleitungen aus.
- Mit einer Steuerleitung ($R/\overline{W}$, was bedeutet: HIGH-aktives Read-Signal) werden die Speicherzellen zum beschreiben frei geschaltet
- Der Spaltenadressdekoder wählt eine der Bitleitungen aus

- Auf dieser Bitleitung wird ein neuer Dateninhalt in die angeschlossene Zelle eingeschrieben
- Bei DRAM-Bausteinen erhalten alle anderen Zellen das Signal verstärkt wieder eingespeichert und somit einen Refresh.

Außerdem verfügt ein

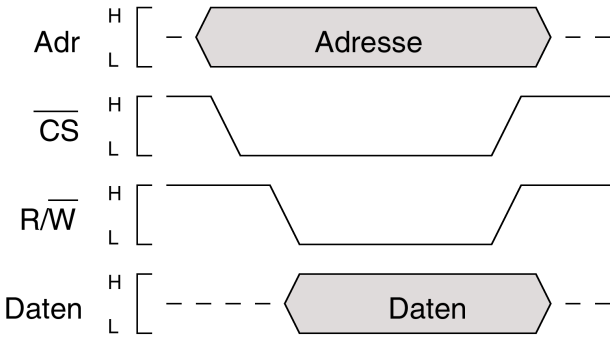


Figure 11.4:
Typischer Ablauf eines Schreibzyklus. Die Daten werden dem Speicherbaustein schon früh im Zyklus zur Verfügung gestellt. Da es viele Adress- und Datenleitungen gibt, ist bei diesen immer durch eine Aufspreizung angedeutet, wann gültige Werte anliegen. $\overline{CS}$ = Chip Select, $R/\overline{W}$ = Read/Write.

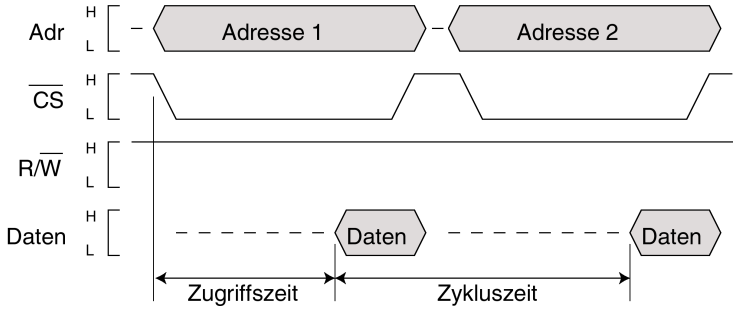


Figure 11.5: Typischer Ablauf zweier aufeinanderfolgender Lesezyklen. Der Speicher stellt die Daten gegen Ende der Zyklen zur Verfügung. Die Zykluszeit ist hier deutlich größer als die Zugriffszeit. das Timing ist genau festgelegt.

Organisation

In vielen Speicherbausteinen ist die Speichermatrix mehrfach vorhanden, dann ergibt sich folgendes:

- Die Adressleitungen sind an alle Speichermatrizen geführt
- Wenn eine Adresse angelegt wird, wird in jeder Matrix die entsprechende Zelle mit dieser Adresse aktiviert
- Es können mehrere Bit gleichzeitig geschrieben oder gelesen werden.
- Beschreibung: *Organisation* des Speicherbausteins

- Beispiel für eine Organisation: 4 Speichermatrizen zu je 8192 Adressen, heißt kurz 8k x 4 .
- Ein 8k x 4 - Baustein hat eine Kapazität von 32 kBit.
- Ein 8k x 4 - Baustein hat 13 Adressleitungen und 4 Datenleitungen

Speicher von Mikroprozessorsystemen sind bis auf Ausnahmen Byte-strukturiert oder Wort-strukturiert. Das bedeutet:

- Man kann an den Speicheradressen kein einzelnes Bit ansprechen, sondern immer nur Gruppen von 8 (bytes), 16 oder 32 Bit (Worte).
- Um eine solchen Speicher aufzubauen, muss man oft mehrere Speicherbausteine parallel schalten. Z.B. ergeben zwei parallel geschaltete Bausteine mit x4-Organisation eine Byte-strukturierten Speicher.

11.2 Read Only Memory (ROM, Festwertspeicher)

Eigenschaften:

- Haben einen festen Dateninhalt
- Sind nicht-flüchtig (Behalten ihre Daten auch ohne Stromversorgung)
- Dateninhalt kann gar nicht oder nicht einfach geändert werden.
- Speichern z.B. die Programme von Embedded systems oder den Bootlader des PC.

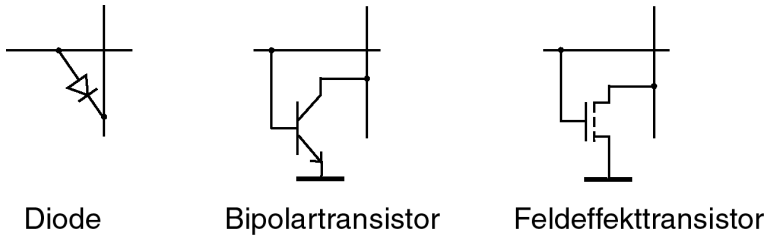


Figure 11.6: In den Speicherzellen von Masken-ROM werden verschiedene Arten von Brücken zwischen Wortleitung und Bitleitung verwendet.

Verschiedene Bauarten ROM

Masken-ROM (MROM) Erhalten ihren Dateninhalt schon bei der Herstellung. Billigste Lösung bei großen Stückzahlen, dieser kann nie mehr geändert werden, daher unflexibel.

Programmable ROM (PROM) Heißen auch OTPROM (One Time Programmable ROM) Dateninhalt kann einmal mit einem Programmiergerät beim Anwender eingeschrieben werden; löschen nicht möglich. Funktionsprinzipien: Fusible Link, AIM oder Floating Gate. Geeignet für Muster und kleine Serien. Unbrauchbar falls falsch beschrieben.

Erasable Programmable ROM (EPROM) Ähnlich wie PROMs, benutzen Floating Gates, können durch einige Minuten UV-Licht (durch Quarzfenster im Gehäuse) wieder gelöscht werden.

Electrical Erasable Programmable ROM (EEPROM) und Flash Weiterentwicklung der EPROMs, erlaubt elektrisches Löschen ohne Entfernung aus der Schaltung. Spezialvariante: Flash, hier werden immer ganze Blöcke gelöscht, lässt sich preiswerter fertigen. Heute das meistbenutzte ROM.

Eigenschaften von Flash-Speichern:

- Spannungswandler und die Programmierlogik können deshalb schon in den Chip integriert werden.
- Im Flash-Speicher werden keine einzelnen Zellen beschrieben, sondern nur Blöcke
- Flash-Speicher halten bis zu 1 Million Schreibvorgänge aus, dann sind sie verbraucht. (Vorsicht bei der Programmierung!)
- Weit verbreitetes Speichermedium, USB-Sticks, Speicherkarten, usw.

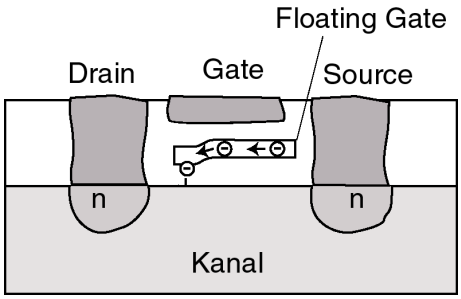


Figure 11.7:
Das Floating Gate eines EEPROM ist über den Drainbereich gezogen. An einer Stelle ist die Isolationsschicht sehr dünn. Dort können Ladungsträger durchtunneln, so dass das Floating Gate elektrisch entladen oder aufgeladen werden kann.

Figure 11.8: Verschiedene Flash-Speicher; Links: Ein USB-Memory-Stick; Mitte: eine Secure-Digital-Card; Rechts: eine CompactFlash-Card

Alle Arten von ROM-Bausteinen sind von großer Bedeutung für eingebettete Systeme, die ja im Regelfall ohne Laufwerke auskommen müssen. Tabelle 11.1 gibt einen Überblick über die ROM-Bausteine und ihre Eigenschaften.

Table 11.1: Eigenschaften und Verwendung von ROM-Bausteinen

Typ	Dauer Schreibvorgang	maximale Anzahl Schreibvorgänge	typische Verwendung
MROM	Monate	1	Ausgereifte Produktion, große Stückzahl
(OT)PROM	Minuten	1	Kleinserie, Vorserie
EPROM	Minuten	bis zu 100	Kundenspezifische Produkte, Entwicklung
EEPROM	Millisekunden	$10^4 - 10^6$	Feldprogrammierbare Systeme
Flash	10 μs	$10^4 - 10^6$	Speichermedien aller Art

11.3 Random Access Memory (RAM)

Eigenschaften aller RAM-Speicher:

- RAM-Bausteine können beliebig oft gelesen und beschrieben werden
- RAM ist ein flüchtiger Speicher
- Schreib- und Lesezugriffe auf RAM gehen schnell
- RAM wird in Computersystemen als Arbeitsspeicher genutzt, alle Variablen und der Stack liegen im RAM
- bei von Neumann-Rechnern liegt auch das Programm im RAM (Bsp. PCs: Beim Programmstart wird der .exe-File ins RAM kopiert)

Statisches RAM (SRAM)

Eigenschaften der statischen RAM-Bausteine:

- Statische RAM-Zellen sind taktgesteuerte D-Flipflops. (Flipflops sind Schaltungen, die man zwischen zwei stabilen Zuständen hin- und her schalten kann.
- Die beiden Flipflop-Zustände stellen '0' und '1' dar.
- Statische RAM-Zellen können schnell ausgelesen und beschrieben (umgeschaltet) werden
- Statische RAM-Zellen sind größer und deshalb auch teurer als DRAM-Zellen.

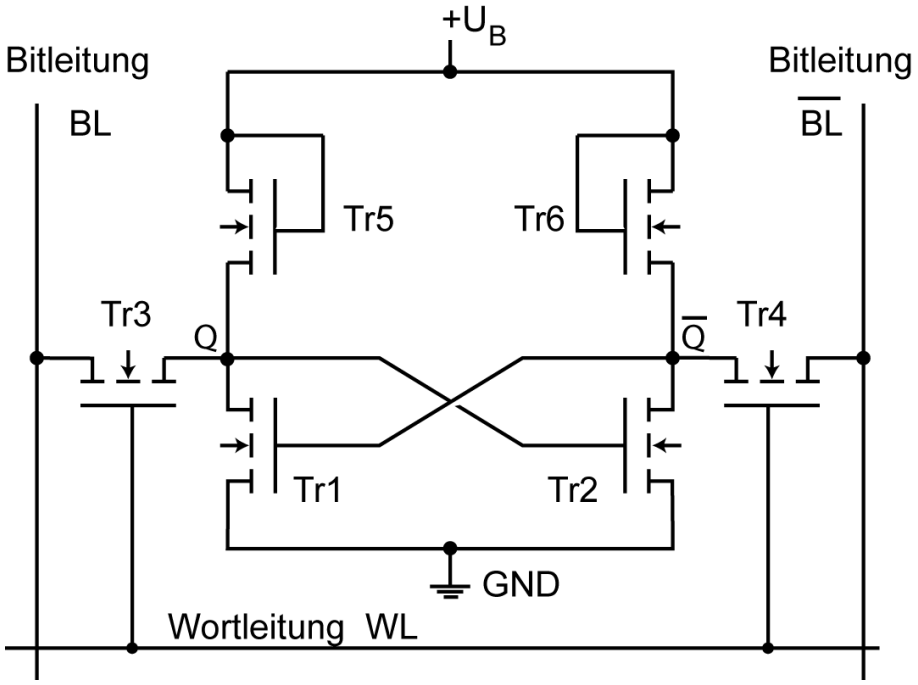


Figure 11.9: Das Flipflop aus zwei NMOS-Transistoren ist der Kern der SRAM-Speicherzelle. Die Wortleitung schaltet über zwei weitere Transistoren die Ein-/Ausgänge auf die Bitleitungen BL und $\overline{BL}$ durch. Über diese erfolgt das Schreiben und Lesen von Daten.

Dynamisches RAM (DRAM)

Eigenschaften der dynamischen RAM-Bausteine:

- Dynamisches RAM (DRAM) lässt sich wesentlich kleiner und billiger herstellen als SRAM und ist daher heute als Hauptspeicher dominierend.
- Bei keinem anderen Speichertyp kann man so viele Zellen auf einem Chip unterbringen (zur Zeit bis 1 GBit)
- DRAMs sind nicht ganz so schnell wie SRAMs
- DRAMs sind Spitzenprodukte der Technik!
- DRAMs werden ständig weiter entwickelt
- DRAM braucht einen regelmäßigen Refresh

Speicherzellen

Die gespeicherte Information wird durch den Ladungszustand eines Kondensators dargestellt:

- Ein ungeladener Kondensator repräsentiert die logische ’0’,
- ein geladener Kondensator repräsentiert die logische ’1’
- (oder umgekehrt)

- Die DRAM-Speicherzelle besteht daher im Wesentlichen aus einem Kondensator, der über einen Tortransistor (Auswahltransistor) mit der Bitleitung verbunden ist (1T1C-Aufbau) (Abb. 11.10).
- Der Tortransistor wird über die Wortleitung angesteuert.
- Das Problem der DRAM-Zelle ist die allmähliche Entladung des Kondensators durch Leckströme. Der Dateninhalt geht also nach einiger Zeit verloren und es ist eine regelmäßige Auffrischung erforderlich, der so genannte *Refresh*.

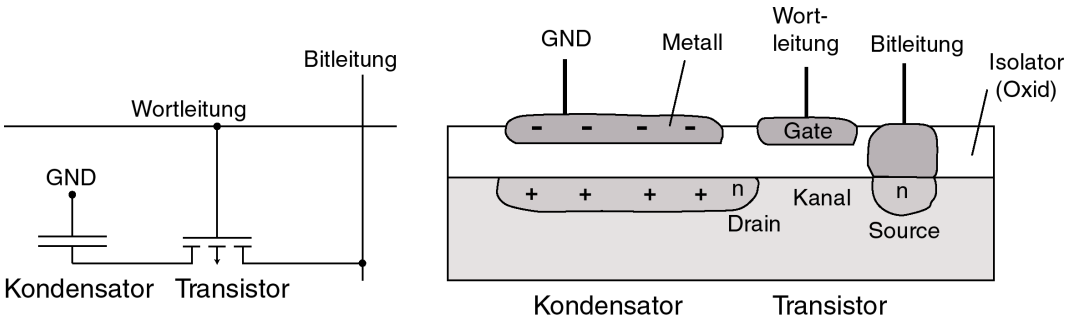


Figure 11.10: Die Zelle eines DRAM besteht aus einem Kondensator und einem Tortransistor. Links Schaltbild, rechts Schichtenaufbau.

Auslesen einer DRAM-Zelle Der Tortransistor der Speicherzelle wird angesteuert (leitend gemacht). Die im Kondensator gespeicherte Ladung gelangt nun auf die Bitleitung und verursacht ein Signal, das nach Verstärkung als Datensignal der Zelle an die Ausgangsleitung des Chips gegeben wird.

Beschreiben einer DRAM-Zelle Das Datensignal erreicht die DRAM-Zelle als elektrisches Potenzial am Dateneingang. Ein Schreib-/Leseverstärker gibt es weiter auf die Bitleitung, während gleichzeitig der Tortransistor der angewählten Zelle angesteuert wird. Wird eine '1' eingeschrieben, so wird die Bitleitung auf die positive Betriebsspannung gelegt und es strömt Ladung in den Kondensator, bis er sich auf Leitungspotenzial aufgeladen hat. Wird dagegen eine '0' eingeschrieben, so liegt die Bitleitung auf $0V$ und der Kondensator wird völlig entladen. Der Tortransistor wird nun gesperrt und die Information ist im Kondensator gespeichert.

11.4 Neuere Speicherbausteine

Ein großer, schneller nicht-flüchtiger Speicher (universal memory) hätte viele Vorteile:

- ein Speicher für alle Aufgaben (Arbeitsspeicher und Massenspeicher)
- Kein Booten und kein Herunterfahren mehr
- Keine Auslagerungsdateien mehr
- Kein Refresh, Stromverbrauch nur noch beim Lesen und Schreiben

Es wird daher intensiv an neuen nicht-flüchtigen Speichermedien geforscht, von denen wir hier drei Richtungen kurz erwähnen wollen.

Magnetoresistives RAM (MRAM) Beruht auf der dauerhaften Magnetisierung einer dünnen hartmagnetischen Schicht, Auslesung mit einem Magnetoresistiven Effekt (der Magnetismus beeinflusst den Widerstand), 10^{15} Schreibzugriffe möglich, Zugriffszeiten von 10 ns, wird schon in Serie gefertigt.

Ferroelektrisches RAM (FRAM) Beruht auf dem ferroelektrischen Effekt, manche Materialien behalten im Feld eines Kondensators eine Polarisation zurück, der Polarisationszustand der Zelle lässt sich durch eine Umpolarisierung messen, damit kann die Zelle ausgelesen werden.

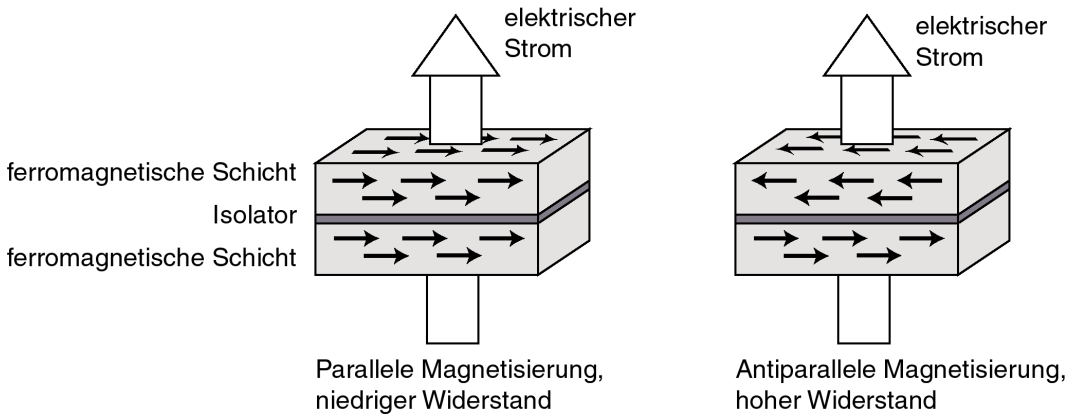


Figure 11.11: Die TMR-Zelle eines MRAM. Der elektrische Widerstand quer durch den Schichtenaufbau wird wesentlich durch die Magnetisierung der beiden Schichten bestimmt.

Übungen und Testfragen

1. Beantworten Sie folgende Fragen zu dem in Abb. 11.12 dargestellten Speicherbaustein:
   a) Welcher Typ Speicher ist dargestellt?
   b) Wie viele Bitleitungen und wie viele Wortleitungen hat er?
   c) Wie viele Bits werden zur Adressierung dieses Bausteines gebraucht und wie werden sie aufgeteilt?
   d) Wie groß ist die Kapazität des Bausteines, wenn er die Organisation x1 hat?
   e) Welchen Inhalt haben die Zellen 14 – 17?

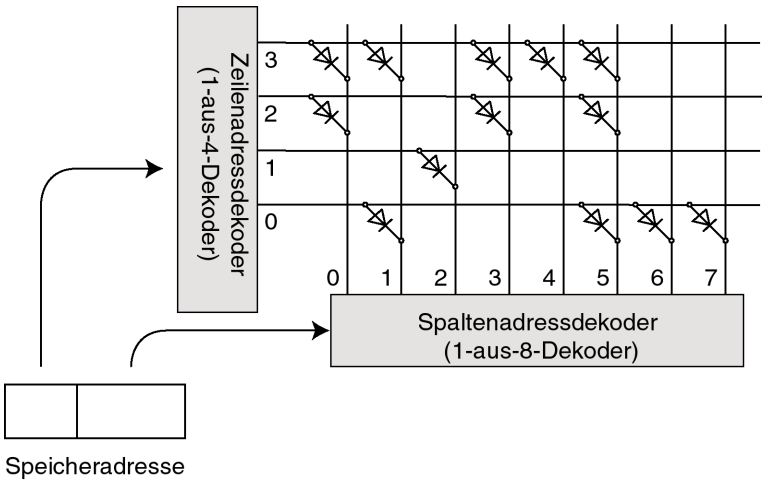


Figure 11.12: Ein Speicherbaustein.

2. Wählen Sie für jede der folgenden Aufgaben einen passenden Speichertyp aus!
   a) Programmspeicherung in einem digitalen Fahrradtachometer nach Abschluss der Tests.
   b) Zwischenspeicherung von Messwerten des Tankgebers in einem Auto, Geberauslesung einmal pro Sekunde.
   c) Programmspeicherung für die Ladeklappensteuerung einer Raumfähre.
   d) Programmspeicherung für Tests an einem Labormuster eines Festplattencontrollers.
3. Nennen Sie drei Methoden, mit denen die Kapazität der Speicherkondensatoren in DRAM-Zellen erhöht wird!
4. Warum gelten SRAM nicht als besonders stromsparende Bausteine?
5. Ein DRAM-Baustein hat einen 4Mx1-Aufbau und eine quadratische Zellenmatrix. Wieviele Adressanschlüsse braucht er mit und ohne Adressmultiplexing? (Adressmultiplexing: Zeilen- und Spaltenadresse werden nacheinander übergeben)
6. Erklären Sie das Funktionsprinzip von MRAM- und FRAM-Chips! Wo liegen ihre Vorteile gegenüber DRAMs?

Antworten

11.5 Übung: "'Landkarte"' des Speichers erstellen

Übung: 11.1 Memory Map zeichnen

Zeichnen Sie eine "'Landkarte"' des Speichers unseres MSP430F2274. Machen Sie die funktionell verschiedenen Bereiche kenntlich und vermerken Sie jeweils Anfangs- und Endadresse.

Übung: 11.2 Speicherort von Programm-Variablen und Code

Betrachten Sie das Übungsprogramm aus dem letzten Teil von Abschnitt 12. Finde Sie heraus, auf welchen Speicherplätzen die Programmvariablen und sonstigen Komponenten während des Programmlaufes liegen. Es geht also um:

- Code
- Stack
- Globale Variablen
- Lokale Variablen
- Lokale static Variablen
- Lokale volatile Variablen
- Arrays
- Konstante Zeichenketten

Zeichnen Sie alle gefundenen Teile in die Memory-Map der vorigen Aufgabe ein!

12 Maschinenbefehlssatz und Maschinencode

12.1 Was passiert bei der Übersetzung? (Einführendes Beispiel)

Beispiel Wir übersetzen das folgende Codestück für einen MSP430 mit der IAR-Entwicklungsgebung:

```
void main(void)
{ int i,j=15;

  for (j=3; j<9; j++) {
    i=3*j - 1;
  }
  if (i==j) P1OUT=0xFE;
}
```

Nach Übersetzen mit der Entwicklungsumgebung und Starten des Debuggers sehen wir:

- Den Quellcode (C-Code) grau als Kommentar eingefügt
- Den erzeugten Maschinencode, der aus Opcodes und Operanden besteht (hexadezimale Schreibweise, zweite Spalte)
- Den erzeugten Assemblercode (Assemblerbefehle plus Operanden (letzte Spalte)
- Die Adressen im Programmspeicher, an denen der Maschinencode abgelegt wurde (erste Spalte)

```
__program_start:
 000200   4031 7000         mov.w   #0x7000,SP
?cstart_call_main:
 000204   12B0 020C         call    #?cstart_end
 000208   12B0 024E         call    #exit
  int i,j=15;
?cstart_end:
main:
 00020C   403F 000F         mov.w   #0xF,R15
  for (j=3; j<9; j++) {
 000210   403F 0003         mov.w   #0x3,R15
 000214   3C06              jmp     0x222
    i=3*j - 1;
 000216   4F0E              mov.w   R15,R14
```

```
 000218   4E0D              mov.w   R14,R13
 00021A   5E0E              rla.w   R14
 00021C   5D0E              add.w   R13,R14
 00021E   533E              add.w   #0xFFFF,R14
  for (j=3; j<9; j++) {
 000220   531F              inc.w   R15
  for (j=3; j<9; j++) {
 000222   903F 0009         cmp.w   #0x9,R15
 000226   3BF7              jl      0x216
  if (i==j) P1OUT=0xFE;
 000228   9F0E              cmp.w   R15,R14
 00022A   2003              jne     0x232
  if (i==j) P1OUT=0xFE;
 00022C   40F2 00FE 0021    mov.b   #0xFE,&P1OUT
}
 000232   4130              ret
__DebugBreak:
 000234   4130              ret
__exit:
 000236   120A              push.w  R10
 000238   8321              decd.w  SP
 00023A   4C0A              mov.w   R12,R10
 00023C   4A81 0000         mov.w   R10,0x0(SP)
 000240   410D              mov.w   SP,R13
 000242   435C              mov.b   #0x1,R12
 000244   12B0 0234         call    #__DebugBreak
 000248   3FF9              jmp     0x23C
_exit:
 00024A   4030 0236         br      #0x236
exit:
 00024E   4030 024A         br      #0x24A
```

Von dem was im Debugger-Fenster zu sehen ist, wird nur der Maschinencode im Programmspeicher abgelegt. Dort wird also stehen:

```
4031 7000 12B0 020C 12B0 024E 403F 000F usw.
```

Dies ist aber nur die hexadezimale Schreibweise für den letztlich binären Code. Diese Schreibweise ist einfach viel kürzer als die binäre, denn wenn wir den Code binär schreiben, steht dort:

```
0100 0000 0011 0001 0111 0000 0000 0000
0001 0010 1011 0000 0000 0010 0000 1100
0001 0010 1011 0000 0000 0010 0100 1110
0100 0000 0011 1111 0000 0000 0000 1111 usw.
```

12.2 Maschinenbefehlssatz

ist ein direktes Abbild aller Operationen, die der Prozessor durchführen kann. (Mit Maschine ist hier der Prozessor gemeint) Die Maschinenbefehle teilen sich in folgende Gruppen auf:

Transportbefehle
Befehle mit denen Daten zwischen Komponenten des Rechnersystems transportiert werden, z.B. Speicher – Speicher, Register – Speicher, Ein-/Ausgabebaustein – Register. Meistens wird dabei eine Kopie des Quelloperanden angelegt. Spezielle Transportbefehle sind die Stackbefehle PUSH und POP.

Arithmetische Befehle
Interpretieren die zu verarbeitenden Bitmuster als Zahlen. Manche Befehle unterscheiden zwischen vorzeichenlosen Zahlen und Zweierkomplement-Zahlen. Typische Befehle: Addition, Subtraktion, Multiplikation, Division, Dekrement, Inkrement und diverse Vergleichsbefehle.

Bitweise Logische Befehle
Hier werden Operanden bitweise durch die logischen Operatoren UND, ODER, exklusives ODER verknüpft; dazu kommt die bitweise Invertierung eines Operanden.

Schiebe- und Rotationsbefehle
Veränderung von Bitmustern durch Schiebe- oder Rotationsbefehle, wie z.B. Schieben nach links, Schieben nach rechts, Rotieren nach links und Rotieren nach rechts. Häufig wird dabei das Carry-Flag einbezogen.

Einzelbitbefehle
Diese Befehle umfassen das Setzen, Verändern oder Abfragen einzelner Bits in Dateneinheiten.

Sprungbefehle
Sie verändern den Inhalt des Programmzählers (PC) und veranlassen dadurch die Fortsetzung des Programms an einer anderen Stelle. Unbedingte Sprungbefehle werden immer ausgeführt, bedingte Sprungbefehle abhängig vom Inhalt des Maschinenstatusregisters (Flags). Spezielle Sprungbefehle sind der Unterprogramm-Aufruf und der Rücksprung aus einem Unterprogramm.

Prozessorsteuerungsbefehle
Jeder Prozessor verfügt über Spezialbefehle, mit denen man die Betriebsart einstellen kann. Dazu zählt z.B. das Freischalten von Interrupts, die Umschaltung auf Einzelschrittbetrieb, die Konfiguration der Speicherverwaltung etc. Oft werden dazu Flags in Steuerregistern gesetzt.

Dies sind nur die wichtigsten Befehlsgruppen, moderne Prozessoren besitzen weitere Befehle, z.B. für die Gleitkommaeinheit. Andererseits sind in einfachen Prozessoren nicht alle aufgeführten Befehlsgruppen vorhanden, z.B. gibt es einfache Prozessoren ohne Einzelbitbefehle.

Der Maschinenbefehlssatz des MSP430

Der Befehlssatz der MSP430-CPU besteht aus 27 elementaren Befehlen (Tabelle 12.1). Zu diesen kommen noch 24 so genannte emulierte Befehle hinzu. Die emulierten Befehle existieren eigentlich nur in der Assemblersprache des Prozessors und werden bei der Übersetzung (Assemblierung) auf elementare Befehle umgesetzt. Die 27 elementaren Befehle bilden drei Gruppen:

- 12 Befehle mit zwei Operanden
- 7 Befehle mit einem Operanden
- 8 Sprungbefehle

Table 12.1: Elementare Befehle des MSP430

ADD(.B) src,dst	Add source to destination
ADDC(.B) src,dst	Add source and Carry to destination
AND(.B) src,dst	AND source and destination
BIC(.B) src,dst	Clear bits in destination
BIS(.B) src,dst	Set bits in destination
BIT(.B) src,dst	Test bits in destination
CALL dst	Call destination
CMP(.B) src,dst	Compare source and destination
DADD(.B) src,dst	Add source and Carryflag decimally to dst
JC/JHS label	Jump if Carryflag set/Jump if higher or same
JEQ/JZ label	Jump if equal/Jump if Zeroflag set
JGE label	Jump if greater or equal
JL label	Jump if less
JMP label	Jump
JN label	Jump if Negativeflag set
JNC/JLO label	Jump if Carryflag not set/Jump if lower
JNE/JNZ label	Jump if not equal/Jump if Zeroflag not set
MOV(.B) src,dst	Move source to destination
PUSH(.B) src	Push source onto stack
RETI	Return from interrupt
RRA(.B) dst	Rotate right arithmetically
RRC(.B) dst	Rotate right through Carry
SUB(.B) src,dst	Subtract source from destination
SUBC(.B) src,dst	Subtract source and not(Carry) from dst
SWPB dst	Swap bytes
SXT dst	Extend sign
XOR(.B) src,dst	Exclusive OR source and destination

Die Befehle, mit denen Operanden bearbeitet werden, wirken standardmäßig auf 16-Bit-Operanden. Durch den Zusatz .B (Byte) kann man bei diese Befehle so einstellen, dass sie 8-Bit-Operanden bearbeiten. Ein Beispiel:

```
ADD #16,R6      ; addiert 16 zu Register 6
ADD.B #16,R6    ; addiert 16 zum niederwertigen Byte von Register 6
```

Binäres Format der Befehle des MSP430

Der Prozessor erkennt weder Hochsprachencode noch Assemblercode sondern nur Maschinencode. Der Maschinencode enthält alle Informationen, die zur Ausführung des Befehles nötig sind in stark codierter kompakter Form, er wird binär (hexadezimal) dargestellt und ist schlecht lesbar. Der Maschinencode enthält binär codierte Maschinenbefehle und Operanden Diesen Code nennt man auch *Operationscode* oder kurz *Opcode* Der Prozessor liest den Maschinencode Byte für Byte ein und führt ihn aus. Da man gelegentlich auch den Maschinencode anschauen und verstehen muss, geben wir hier eine kurzen Überblick über den Maschinencode des MSP430.

Die Hauptbestandteile der Opcodes sind

- Bitmuster, die den auszuführenden Befehl beschreiben (die eigentlichen Opcodes)
- Angaben zu den benutzten Registern
- Angaben zur Adressierung der benutzten Speicheradressen, (Adressing modes)
- Angaben ob der Befehl auf 8 oder 16 Bit Breite ausgeführt wird

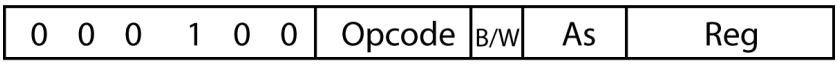
Die beiden letzten Informationen werden als so genannte Steuerbits in die Befehle eingefügt:

Steuerbit	Bedeutung	Codierung
B/W	Byte /Word	1=Byte, 0=Word
As	Adressing mode source	00=Register direkt (Inhalt) 01= indexed Register indirekt 10 = Register indirekt 11 = Register indirekt mit Autoinkrement
Ad	Adressing mode destination	0 = Register direkt (Inhalt) 1 = indexed Register indirekt
Reg	Register	Nummer des Registers als 4-Bit-Zahl (0 - 15)

Figure 12.1: Bedeutung und Codierung der Steuerbits im Maschinencode des MSP430.

Die Bedeutung der Steuerbits hängt vom Befehl ab, so kann z.B. As=11 für Autoinkrement stehen oder für einen unmittelbar im Code folgenden Operanden (immediate operand)

Befehle mit einem Operanden



Mnemonic	Befehl	Opcode
RRC	Rotate right	000
SWPB	Swap bytes	001
RRA	Rotate right arithmetic	010
SXT	Sign extend byte to word	011
PUSH	Push Value on Stack	100
CALL	Call Subroutine	101
RETI	Return from Interrupt	110

Figure 12.2: Maschinencode der Befehle des MSP430 mit einem Operanden.

Beispiel:

Opcode des Befehles push.w R9

Variabler Teil des Opcodes = 100 für push
B/W = 0 für word (16 Bit)
As = 00 für Register direkt
Reg = 1001 für Reg.9

Ergebnis: 1209h

Figure 12.3: Maschinencode für den Befehl push.w R9 .

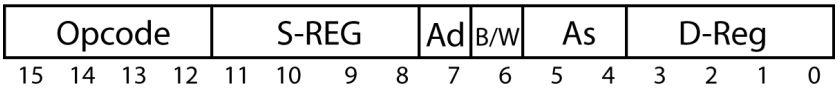
Übung: 12.1 Befehle mit einem Operanden kodieren/dekodieren

a)Codieren Sie den Befehl "'swap Bytes in R12"' (swpb R12)
b)Dekodieren Sie den Befehlscode 118Dh.

Übung: 12.2 Verständnisfrage

Warum kann der Registerstapel nicht mehr als 16 Register haben?

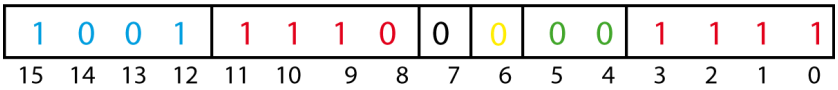
Befehle mit zwei Operanden



Mnemonic	Befehl	Opcode
MOV	Move	0100
ADD	Add	0101
ADDC	Add with Carry	0110
SUBC	Subtract with Carry	0111
SUB	Subtract	1000
CMP	Compare	1001
DADD	Decimally Add	1010
BIT	Bit Test	1011
BIC	Bit Clear	1100
BIS	Bit Set	1101
XOR	Logical exclusive OR	1110
AND	Logical AND	1111

Figure 12.4: Maschinencode der Befehle des MSP430 mit zwei Operanden.

Opcode von cmp.w R14 , R15



Opcode: = 1001 für compare
Source reg. = 1110 für R14
Ad = 0 für register mode
B/W = 0 für word
As = 00 für register mode
Destination reg = 1111 für R15

Ergebnis: 9E0F

Figure 12.5: Maschinencode des Befehls cmp.w R14,R15.

Übung: 12.3 Maschinencode mit zwei Operanden

a) Codieren Sie den Befehl "'add R12 to R11"' (add R12,R11)
b) Dekodieren Sie den Opcode 8A09h

Übung: 12.4 Maschinencode dekodieren

Dekodieren Sie den Maschinencode 403Ch 0007h

Sprungbefehle

0 0 1	Condition	Offset (Abstand) zum Sprungziel
15 14 13	12 11 10	9 8 7 6 5 4 3 2 1 0

Mnemonic	Befehl	Opcode
JNE/JNZ	Jump if not equal/zero	000
JEQ/JZ	Jump if equal/zero	001
JNC/JLO	Jump if no carry / lower	010
JC/JHS	Jump if carry / Higher or same	011
JN	Jump if negative	100
JGE	Jump if greater or equal	101
JL	Jump if less	110
JMP	Jump always (unconditionally)	111

Figure 12.6: Maschinencode der Sprungbefehle des MSP430.

Beispiel:

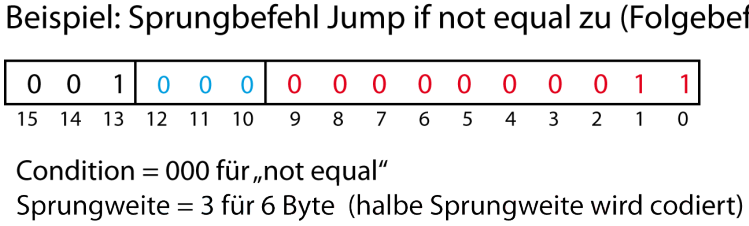


Figure 12.7: Maschinencode des Sprungbefehls jump if not equal (+6 Byte).

Übung: 12.5 Sprungbefehle kodieren/dekodieren

a)Codieren Sie den Befehl "'jump +10 Byte"' (jmp +10)
b)Dekodieren Sie den Befehlscode 27FBh.

12.3 Der Aufbau des Maschinencodes und seine Ausführung

- Bei vielen Maschinenbefehlen gehört zum Opcode einer oder mehrere Operanden, mit weiteren Angaben zur Ausführung
- Die Operanden folgen im Speicher dem Opcode
- Es gibt auch Opcodes, die keine Operanden brauchen, z.B. "'Lösche das Caryflag"'
- Die gemischte Folge aus Opcodes und Operanden heißt *Maschinencode*
- Der Maschinencode stellt das ausführbare Programm dar. (Bei Windows: .exe-File)
- Der Prozessor liest den Maschinencode Byte für Byte ein und führt ihn aus.

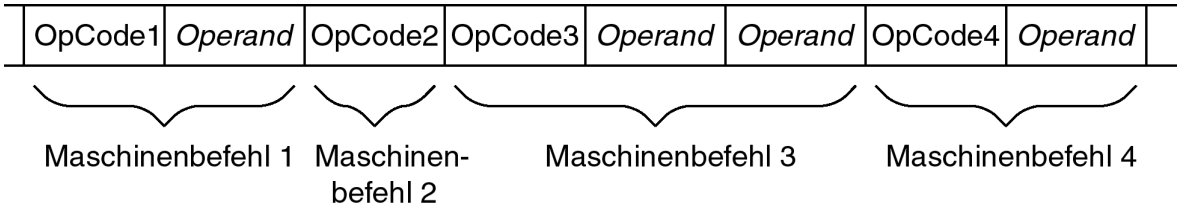


Figure 12.8: Maschinencode besteht aus einer Folge von OpCodes und zugehörigen Operanden.

Jeder Prozessor verfügt über einen *Programmzähler*, der die Adresse des nächsten auszuführenden Befehles enthält. *Die grundsätzliche Funktionsweise eines Mikrorechnersystems ist nichts weiter als eine endlose Wiederholung einer kleinen Sequenz, bei der der anstehende Maschinenbefehl ausgeführt und anschließend der Programmzähler erhöht wird.* Dazu kommt das Auslesen der Operanden, die ja zwischen den Opcodes stehen. Im Einzelnen:

- Maschinencode wird in den Programmspeicher geladen (oder steht fest dort im Flash)
- Der nächste auszuführende Opcode aus dem Programmspeicher gelesen (Opcode-Fetch)
- Der Opcode wird gefunden mit Hilfe des *Programmzählers* (Program Counter, PC), ein spezielles Register, das immer die Adresse des nächsten einzulesenden Bytes im Maschinencode enthält
- Befehl wird dekodiert, d.h. bitweise mit bekannten Mustern verglichen, um seine Bedeutung herauszufinden
- Falls der Opcode aussagt, dass zu diesem Befehl auch Operanden gehören, wird der Programmzähler inkrementiert, um den ersten Operanden auf dem nachfolgenden Speicherplatz zu lesen
- Dies wird so lange wiederholt, bis alle Operanden gelesen sind.
- Befehl wird ausgeführt.
- Währenddessen wird der Programmzähler ein weiteres Mal inkrementiert und zeigt nun auf den Opcode des nächsten Befehls.

- Der Befehlszyklus beginnt von vorne mit dem Laden des nächsten Opcodes

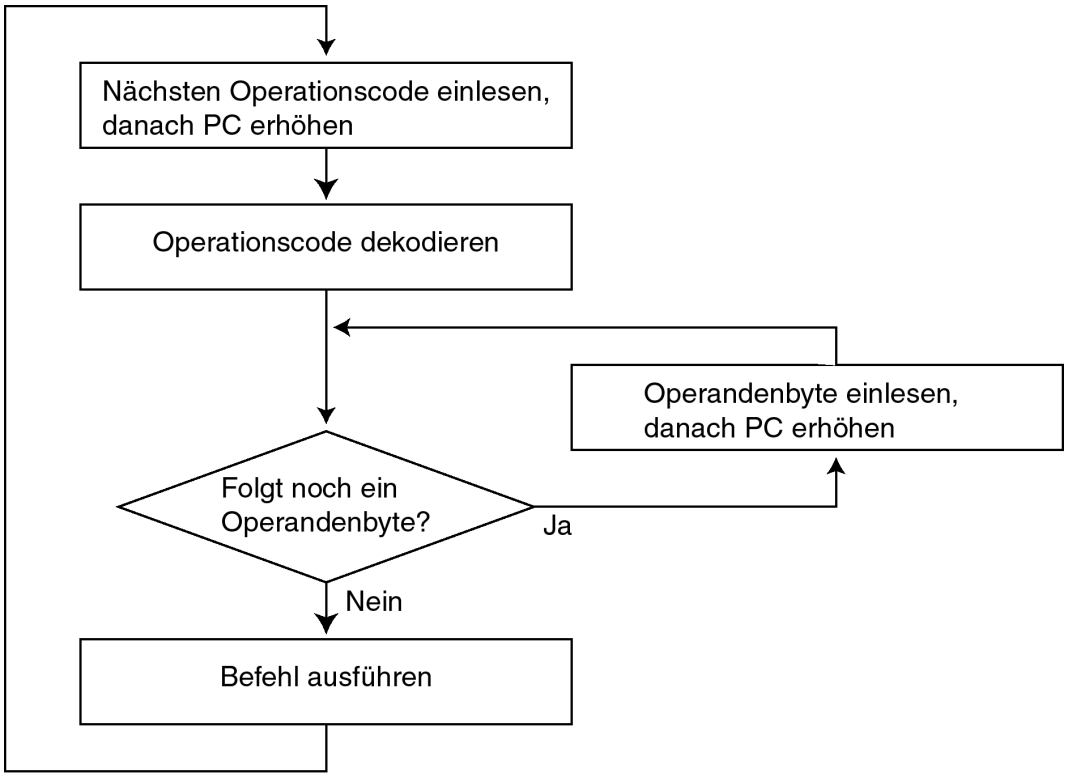


Figure 12.9: Im Befehlszyklus werden zunächst der Befehl und die eventuell vorhandenen Operanden eingelesen, erst dann kann der Befehl ausgeführt werden.

Sprünge werden einfach realisiert, indem der Programmzähler einen neuen Wert erhält. Damit werden Wiederholungsschleifen und Verzweigungen realisiert.

- Programmzähler wird mit einem kleineren Wert überschrieben → Rückwärtssprung → Code wird erneut ausgeführt → Schleife
- Programmzähler wird mit einem größeren Wert überschrieben → Vorwärtssprung → Code wird übersprungen → Bedingte Ausführung ("'if"')

Beispiel:
Ein Befehl zum Laden von Daten aus dem Speicher mit einer nachfolgenden 16-Bit-Adresse ist ein Maschinenbefehl mit zwei Operandenbytes. (Abb. 12.10).

Wir halten fest:

- Durch das Überschreiben des Programmzählers können Sprünge im Programm realisiert werden, mit den Sprüngen werden wiederum Verzweigungen und Wiederholungen aufgebaut.
- Jeder Befehlszyklus besteht intern aus vielen Einzelschritten, die von der Prozessorhardware selbständig ausgeführt werden.
- Jedes Lesen aus dem Speicher oder Schreiben in den Speicher ist ein Buszugriff (Buszyklus)
- Wie man sieht, kann ein Befehlszyklus mehrere Buszyklen umfassen.

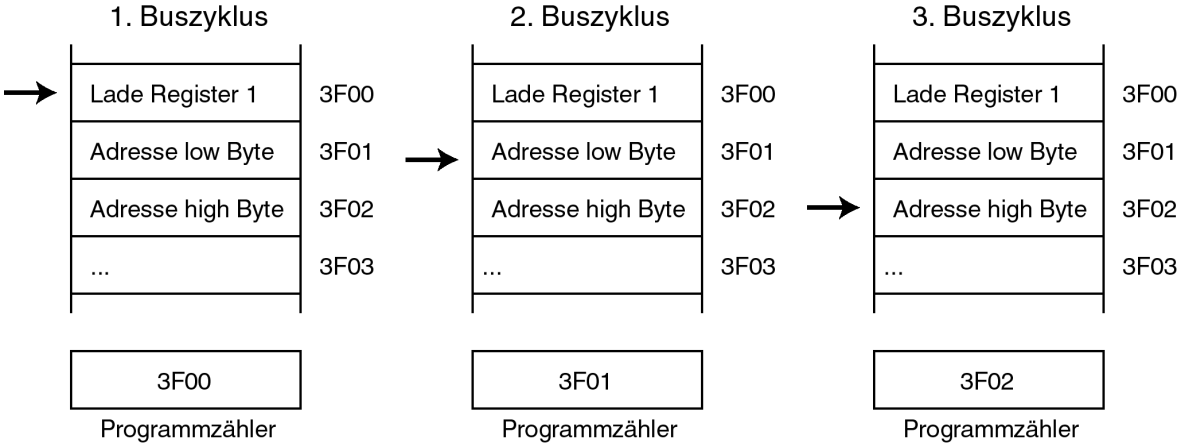


Figure 12.10: Ein Befehl mit zwei Operanden-Bytes wird eingelesen. Der Befehlszyklus besteht hier aus drei Buszyklen: Opcode lesen, erstes Operanden-Byte lesen, zweites Operanden-Byte lesen.

12.4 Maschinencode verstehen an einem größeren Beispiel

Wir benutzen als Lernobjekt folgendes Programm:

```
#include <msp430x22x2.h>

// Funktionen
int rechne(int a, int b){
int summe;
summe = a+b;
return (9*summe - b);
}

void Warteschleife(unsigned int loops) {
unsigned int i;       // Eine Variable wird heruntergezählt bis auf 0.
  for (i = loops; i > 0; i--);
}

int globalVar;

int main(void) {
  char *string1="Hallo Welt!";
  int Zahlen[10];
  int i,j;
  static int k;

  WDTCTL = WDTPW + WDTHOLD;       // Watchdog Timer anhalten

  globalVar=80;

  i=3;
```

```
  j=4;
  k=5;
  i=k+j;

  k=rechne(i,j);
  Zahlen[2]=k;

  k=i*j;
  k=Zahlen[2]+globalVar;

  // Hardware-Initialisierung
  P1DIR = 0xFF;                    // P1.0-P1.7 ist Ausgabeport
  P1OUT = (k & 0xFF);

  Warteschleife(100);

  while(1)  {    // Endlosschleife
    __low_power_mode_3();       // low power mode 3
    __no_operation();           // Required only for C-spy
  }
}
```

Hier der Link auf den Democode:

Übung: 12.6 Analyse des Assembler- und Maschinencodes

Übersetzen Sie das oben stehende kleine Programm und analysieren Sie den erzeugten Code. Benutzen Sie eine Entwicklungsumgebung wie die IAR Embedded Workbench und den MSP430 Family User's Guide.

Bei der Bearbeitung dieser Übung sollten die folgenden Fragen beantwortet werden:

1. In welche Assembler- bzw. Maschinenbefehle werden die Zuweisungen aus C übersetzt?
2. In welche Assembler- bzw. Maschinenbefehle werden die Funktionsaufrufe aus C übersetzt?
3. Wie ist die Warteschleife auf Maschinenebene realisiert?
4. Wie wird der Mikrocontroller auf Maschinenebene in den Low-Power-Mode 3 umgeschaltet?
5. Was wird aus __no_operation(); auf der Assemblerebene und auf der Maschinenebene?
6. Wo stehen die Operanden der Befehle "'i=3"', "'j=4"' und "'k=5"' im Assembler- und im Maschinencode?
7. Können Sie erkennen, wo im Maschinencode die Nummer des angesprochenen Registers steht?
8. Mit welchem Maschinenbefehl wird die Multiplikation "'k=i*j"'ausgeführt?
9. Wie wird die Multiplikation "'summe*9"' in der Funktion "'rechne"' ausgeführt?

Hinweis: Wenn die IAR-Entwicklungsumgebung verfügbar ist, übersetzen Sie das Programm, laden Sie es in den Debugger (Simulation, Device=MSP430F2272), öffnen Sie die Disassembly-View und gehen Sie es schrittweise durch, um die unten stehenden Fragen zu beantworten.

Der erzeugte Maschinencode wird natürlich vom Übersetzer auf die benutzte Hardwareplattform, den Device, angepasst. Die folgende Übung macht das deutlich.

Übung: 12.7 Codeerzeugung bei geänderter Hardwareplattform

Wie ändert sich der erzeugte Maschinencode, wenn das gleiche Programm für den MSP430F449 übersetzt wird.

Hinweis: Wenn die Entwicklungsumgebung verfügbar ist, stellen Sie unter Project/Options/Device um auf den Mikrocontrollertyp MSP430F449 um, laden Sie das Programm aus der vorigen Aufgabe wieder in den Debugger (Simulation) und schauen Sie erneut in den Maschinencode. Was hat sich geändert?

13 Mikrocontroller: Bausteine für die Betriebssicherheit

Mikrocontroller werden oft in Steuerungen eingesetzt, Fehlfunktionen haben evtl. gravierende Folgen. (z.B. bei großen Maschinen) Daher sind sie meistens mit speziellen Bausteinen zur Verbesserung der Betriebssicherheit ausgerüstet.

13.1 Watchdog-Timer

Ein Mikrocontroller könnte durch einen versteckten Programmfehler oder durch umgebungsbedingte Veränderungen von Speicher- oder Registerinhalten, z.B. Störsignale, in eine Endlosschleife geraten. Damit fällt er praktisch aus und die Steuerung ist blockiert. Diese gefährliche Situation soll ein *Watchdog-Timer* (WDT) vermeiden.

Funktionsweise Watchdog-Timer:

- WDT ist ein freilaufender Zähler, der bei Überlauf einen Reset des Mikrocontrollers auslöst.
- Normaler Programmablauf. WDT wird regelmäßig von Software zurückgesetzt, um WDT-Reset zu vermeiden.
- Fehlerfall Endlosschleife: Zurücksetzen des Watchdog-Timer findet nicht mehr statt, Reset findet statt, System fährt neu hoch und gerät (mit Glück) nicht mehr in die Endlosschleife, d.h. arbeitet wieder
- Ausnahme: WDT-zurücksetzen liegt in der Endlosschleife (Pech gehabt ...)

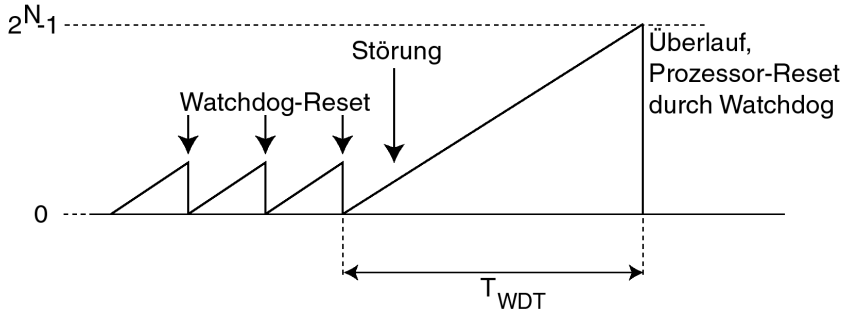


Figure 13.1: Ein Watchdog-Timer muss regelmäßig vom Programm zurückgesetzt werden, sonst löst er ein Reset aus.

Spezailvariante: Ein *Oscillator-Watchdog* überwacht den Oszillator, bei Ausfall schaltet er auf eigenen Oszillatortakt um und löst Reset aus.

Fallbeispiel: Der Watchdog Timer+ des MSP430

- aufwärts laufender 16-Bit-Zähler
- bei Erreichen eines eingestellten Endwertes auslösen eines Reset (genauer: Power Up Clear)
- Muss deshalb vorher vom Programm auf 0 zurückgesetzt werden, das geschieht durch Zugriff auf das Bit WDTCNTCL (Watchdog Timer Counter Clear) im Watchdog Timer Control Register
- Endwert bestimmt die Länge des WDT-Intervalls und ist einstellbar auf 32768 (Voreinstellung), 8192, 512 oder 64.
- Das Steuerregister ist gegen irrtümliche Zugriffe geschützt: Schreibzugriff auf das Steuerregister nur möglich mit "'Passwort"' 5Ah im oberen Byte
- Der WDT+ kann mit SMCLCK oder ACLCK getaktet werden
- Wenn keine Watchdog-Funktion nötig, Verwendung als normaler Intervall-Timer möglich.

Das Watchdog Timer+ Control Register (WDTCTL) enthält unter anderem folgende Bitfelder: [1]

WDTPW WDT Password, 8-Bit-Passwort, nicht auslesbar

WDTHOLD WDTHold, Anhalten des WDT+

WDTTMSEL WDT Mode Select: Watchdog oder intervall timer

WDTCNTCL WDT Counter Clear, setzt den Zähler auf 0 zurück

WDTSSEL WDT Source Select, wählt ACLK oder SMCLK

WDTIS WDT Interval Select, Legt das Zählintervall fest auf 32768, 8192, 512 oder 64 Takte

13.2 Brown-Out-Protection

Unter "'Brown-Out"' versteht man einen kurzzeitigen Einbruch der Versorgungsspannung unter den erlaubten Minimalwert. Folgen:

- Der Brown-Out löst kein Power-on-Reset aus, da Spannung nicht niedrig genug
- Inhalt von Speicherzellen und Registern evtl. verändert
- Kritische Situation, Fehlfunktion möglich!
- Brown-Out-Protection löst Reset aus, System wird neu gebootet, wieder konsistent

[1] Zusätzlich wird auch die Funktion des RST/NMI-Eingangs vom WDTCTL gesteuert.

14 Energieeffizienz von Mikroprozessoren

14.1 Was ist Energieeffizienz und warum wird sie gebraucht?

Wir meinen mit Energieeffizienz, dass die elektrische Leistungsaufnahme (der "'Stromverbrauch"') in einem möglichst guten Verhältnis zur aktuellen, tatsächlich erbrachten Rechnerleistung steht. Warum Energieeffizienz?

Betriebszeit akkubetriebener Geräte Läuft das Notebook 5 Stunden oder 30 Minuten? Ähnlich Handys, GPS-Empfänger usw.

Kühlungsaufwand und die damit verbundene Geräuschentwicklung Die ersten PC-Prozessoren liefen noch ganz ohne Kühler, danach war ein passiver Kühlkörper nötig, später musste auf den CPU-Kühler noch ein eigener Lüfter gesetzt werden; auch Grafikkarten sind längst aktiv gekühlt

Betriebskosten Ein PC mit 80 Watt Leistung, der durchgehend läuft, verursacht in einem Jahr schon 140,-Euro Kosten. Bei Rechenzentren sind die Energiekosten der größte Teil der Gesamtkosten!

Klimaschutz Der gleiche PC setzt im Jahr 432 kg CO_2 frei, dazu müsste ein Auto 3200 km fahren!

14.2 Leistungsaufnahme von integrierten Schaltkreisen

Leistungsaufnahme von integrierten Schaltkreisen mit Feldeffekttransistoren (FETs) (heute dominierende Bauweise):

- Statische Verlustleistung, verursacht durch Leckströme an den FETs
- Dynamische Verlustleistung, verursacht durch die Umladung der unvermeidlichen parasitären Kapazitäten auf dem CMOS-Chip.

$$P_d = fCU^2 \tag{14.1}$$

Dabei ist

P_d	dynamischer Anteil der Leistungsaufnahme
f	Arbeitsfrequenz
C	Gesamtkapazität der Feldeffekttransistoren im Schaltkreis
U	Betriebsspannung

Die gesamte Leistungsaufnahme eines Schaltkreises mit FETs ergibt sich mit der statischen Leistungsaufnahme P_s als:

$$P = P_s + P_d = P_s + fCU^2 \tag{14.2}$$

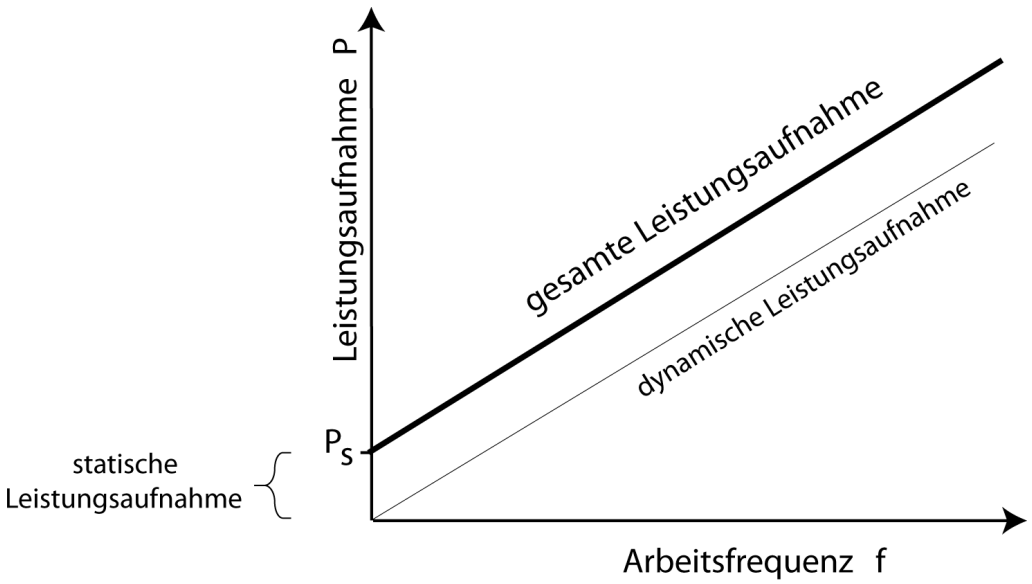


Figure 14.1: Die Leistungsaufnahme eines Schaltkreises mit FETs setzt sich aus einem statischen und einem dynamischen Anteil zusammen.

Praktisches Beispiel: Aus dem Datenblatt des MSP430 geht hervor, wie der Versorgungsstrom sowohl mit der Spannung (V_{cc}) als auch der Arbeitsfrequenz (f_{DCO}) linear ansteigt. (Abb. 14.2) Im linken Teil der Abbildung ist zu sehen, dass der Strom I annähernd linear mit U ansteigt. Da die Leistungsaufnahme $P = UI$ ist, bedeutet dies, dass P proportional zu U^2 ansteigt – wie man es nach Gleichung 14.1 erwartet.

Verminderung der Leistungsaufnahme

Um die dynamische Leistungsaufnahme zu reduzieren kann man nach Formel 14.2 drei Wege gehen und in der Praxis werden tatsächlich alle drei Möglichkeiten genutzt:

- Die Arbeitsfrequenz f absenken
- Die gesamte Kapazität C vermindern
- Die Betriebsspannung U absenken

Die Arbeitsfrequenz wird heute immer wenn es möglich ist, abgesenkt. Bei PCs geschieht das in vielen Stufen nach dem Standard *Advanced Control and Power Interface* (*ACPI*). Bei Mikrocontrollern wird im Low-Power-Mode die Frequenz auf Null gesenkt und im Active Mode wieder auf den Normalwert gesetzt.

Betriebsspannung U wirkt quadratisch auf Leistungsaufnahme, Absenkung besonders wirksam. Betriebsspannungen der Prozessoren: 5V – 3.3V – 2.8V – 2V – ≈1V; ähnlich bei Speichern;

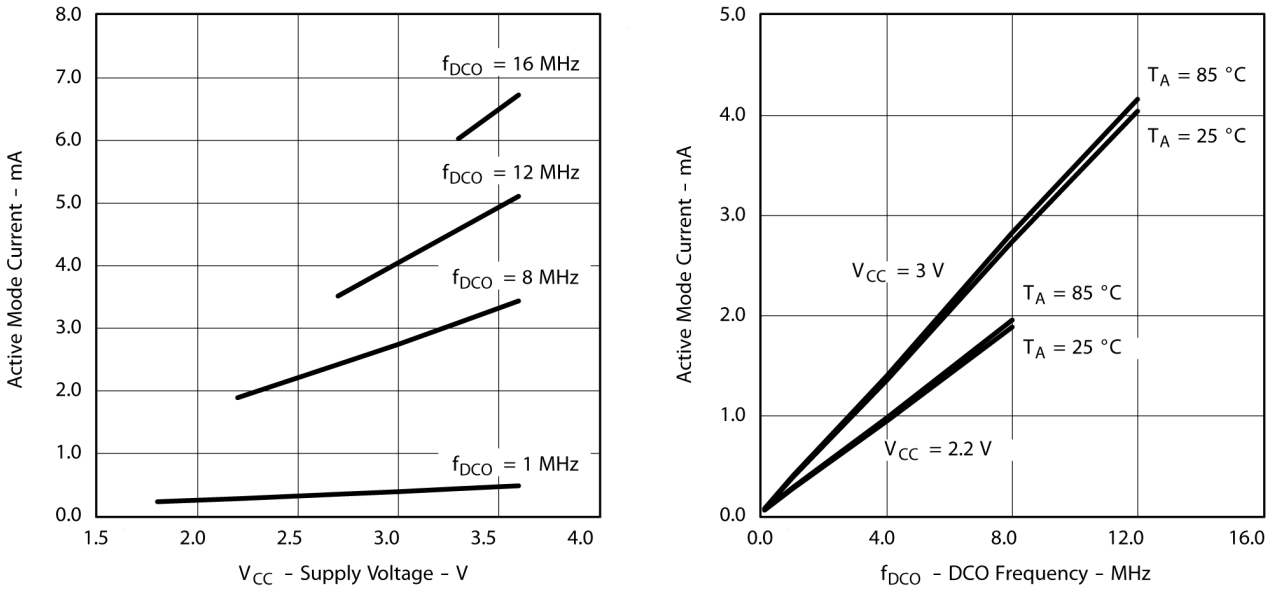


Figure 14.2: Der Versorgungsstrom eines Mikroprozessors steigt mit der Versorgungsspannung an (Bild links). Bei konstanter Versorgungsspannung steigt der Versorgungsstrom linear mit Arbeitsfrequenz an (Bild rechts). (Aus dem Datenblatt des TI MSP430 x22x2 mit freundlicher Genehmigung von Texas Instruments. [47])

Die Gesamtkapazität C ergibt sich wie folgt:

$$C_{gesamt} = C_{fet} \cdot N_{Fet} + C_{leitung} \cdot N_{Leitung}$$

Im Betrieb lässt sich C nur durch Abschaltung von Teilen des Chips, z. B. Bussen, vermindern. Verfahren zur Reduzierung von C in Tab. 14.2.

Verfahrensbezeichnung	**Funktionsweise**
Optimierung des Clock-Trees und Clock-Gating	Von mehreren Taktleitungen werden die unbenutzten zeitweise abgeschaltet
Multi-Thresholding (Multi-V_{Th})	Mehrere Bereiche, in denen Logikbausteine unterschiedliche Transistoren nutzen: Solche mit hohen und solche mit niedrigen Schaltschwellenspannungen
Mehrere Versorgungsspannungen	Jeder Funktionsblock erhält die optimale (möglichst niedrige) Versorgungsspannung
Dynamische Spannungs- und Frequenzanpassung	In ausgewählten Bereichen des Chips wird während des Betriebs Versorgungsspannung und Frequenz an die aktuell geforderte Leistung angepasst.
Power-Shutoff	Nicht verwendete Funktionsblöcke werden abgeschaltet

Table 14.1: Einige Verfahren zu Verminderung der Leistungsaufnahme von Mikroprozessoren.

Energieeffizienz liegt in der Verantwortung der Software-Entwickler. Sie müssen ein gutes Verständnis der Stromspar-Mechanismen haben!

14.3 Energie-Effizienz am Beispiel des MSP430-Mikrocontrollers

- Der MSP430 ist als Low-Power-Mikrocontroller entworfen worden. (Hersteller: Ultra Low Power MCU)
- 4 Low-Power-Modes
- So sind die Baugruppen der Peripherie, manchmal sogar in Teilen, einzeln abschaltbar um Strom zu sparen.
- Das macht die Aktivierung der Peripheriegruppen in der Software etwas komplizierter, man hat sich aber schnell daran gewöhnt.
- Mehrere taktsignale zur Wahl für Peripheriegruppen

Versorgungsspannung 1.8 V und 3.6 V, beeinflusst Leistungsaufnahme quadratisch!

Arbeitsfrequenz (1) zwischen 12 kHz und 16 Mhz, mit dem Digitally Controlled Oscillator (DCO) die sogar per Software einstellbar zwischen 60 kHz und 12 MHz! (allerdings ohne Quarzstabilisierung)

Arbeitsfrequenz (2) 3 verschiedene Taktsignale auf dem MSP430, variabel aus vier verschiedenen Oszillatoren erzeugt, Wahlmöglichkeit bei Peripherie

Absenkung von C_{gesamt} In fast allen Mikrocontrollern ist es möglich Teile des Chips abzuschalten. (senkt auch die statische Verlustleistung)

Taktsignale:

MCLK Masterclock, versorgt CPU

ACLK Auxiliary Clock, quarzstabilisiert, für Peripherie

SMCLK Sub main Clock

Der Anwender kann für viele Peripheriekomponenten einen der drei Takte auswählen.

Sehr wichtig für die Programmierung: Es gibt insgesamt 5 Low-Power-Betriebsarten: (Betriebsstrom bei 1 MHz und 3V genannt)

aktive Betriebsart (Active Mode) Alle Taktsignale aktiv, CPU arbeitet (ca. 300 μA)

Low Power Mode 0 (LPM0) CPU und MCLK abgeschaltet, ACLK und SMCLK aktiv (ca. 90 μA)

Low Power Mode 1 (LPM1) CPU, MCLK und abgeschaltet, ACLK und SMCLK aktiv (DCO-Oszillator abgeschaltet, wenn nicht gebraucht)

Low Power Mode 2 (LPM2) CPU, MCLK, SMCLK abgeschaltet, DC-Generator des DCO bleibt aktiv (ca. $25\,\mu A$)

Low Power Mode 3 (LPM3) CPU, MCLK, SMCLK, DC-Generator des DCO abgeschaltet, ACLK bleibt aktiv (ca. $1\,\mu A$)

Low Power Mode 4 (LPM4) CPU, MCLK, SMCLK und ACLK abgeschaltet, DCO und Kristall-Oszillator gestoppt. (ca. $0.1\,\mu A$)

Übung: 14.1 Verlustleistung von Prozessoren

a) Ein Mikroprozessor nimmt bei 2.4 Ghz Takt 42 Watt Leistung auf. Wieviel Leistung wird durchschnittlich aufgenommen, wenn in einem Beobachtungszeitraum der Prozessor 50 % im Haltzustand ist (f=0), 40 % mit 0.8 Ghz und 10 % mit 2.4 MHz läuft?
b) Ein TI MSP430 soll mit 2.5 V Versorgungsspannung bei einem Takt von 8 MHz betrieben werden. In bestimmten rechenintensiven Situationen soll der Takt mit dem DCO auf 12 MHz erhöht werden. Wo liegt das Problem bei dieser Planung?

Verwendung der Low-Power-Modes Sobald möglich (Wartezustand), in einen der Low Power Modes wechseln. Achtung: Sicherstellen, dass der Controller schnell genug in den Active Mode kommt! Eine sehr niedrige durchschnittliche Leistungsaufnahme ergibt sich, wenn der Chip ganz wenig im Active Mode und statt dessen meistens in einem der Low Power Modes ist.

Es gibt weitere Regeln für eine energieeffiziente Programmierung:

- nicht benötigte Teile abschalten, wann immer möglich
- So weit wie möglich, Aufgaben durch Hardware statt durch Software erledigen lassen. Ein PWM-Signal sollte z.B durch einen Timer mit PWM-Kanal erzeugt werden.
- Möglichst Interrupts plus Low Power Mode statt Pin- oder Flag-Polling verwenden
- Vorberechnete Tabellenwerte statt aufwändiger Berechnungsroutinen verwenden.
- Häufige Aufrufe von Unterprogrammen vermeiden und stattdessen den Programmfluss einfacher steuern, z. B. mit berechneten Sprungzielen

Diese Regeln decken sich nicht alle mit den Grundsätzen der üblichen Softwaretechnik, sind aber unter dem Gesichtspunkt der Energieeffizienz gerechtfertigt.

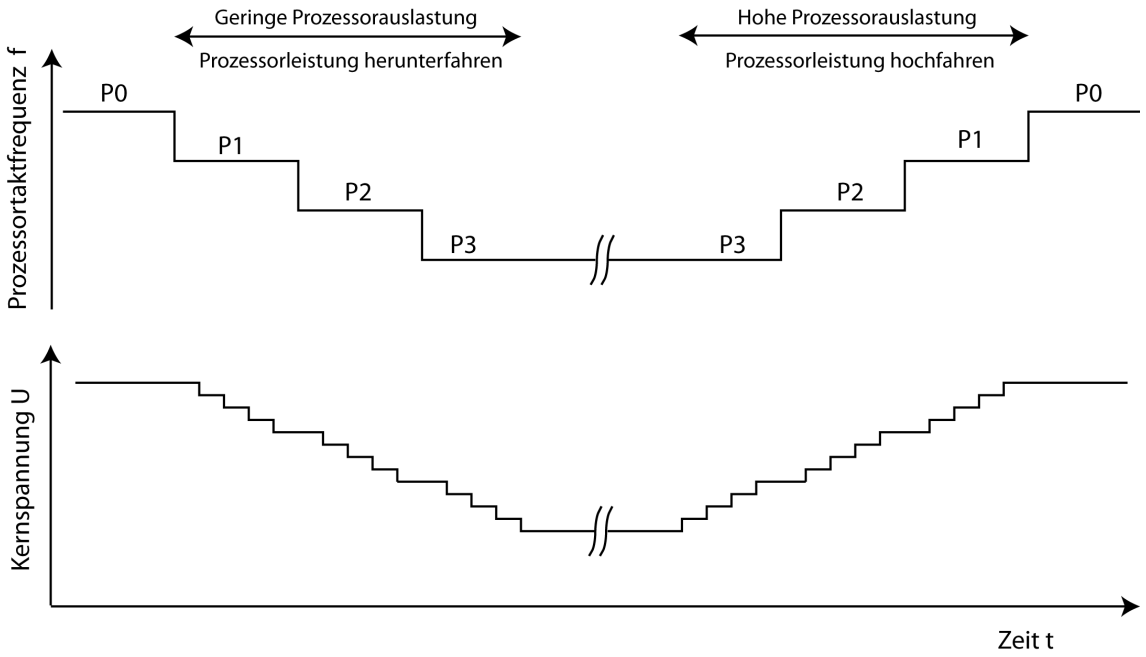


Figure 14.3: Anpassung von Taktfrequenz und Kernspannung beim Speedstep-Verfahren von Intel. In diesem Beispiel wird herunter gefahren bis zum Zustand P3

Übung: 14.2 Testfragen

1. Wie lautet die Formel für die Leistungsaufnahme elektronischer Schaltkreise?
2. Wie lange könnte man ein Desktop-System mit einer Leistung von 250 W aus einem Akku mit 11V Spannung und einer Kapazität von 4000 mAh betreiben?

15 Der Umgang mit gemeinsamen Daten

15.1 Was sind gemeinsame Daten (Shared Data)?

In einem eingebetteten System finden wir üblicherweise folgende Situation:

- Einer oder mehrere Interrupts sind aktiviert
- Nicht die ganze Arbeit wird in den Interrupt-Service-Routinen gemacht
- Es muss Information zwischen dem Hauptprogramm und den Interrupt-Service-Routinen ausgetauscht werden
- Der Informationsaustausch findet über globale gemeinsame Variablen (Shared Data) statt
- Die Programmierung ist nebenläufig (mehrere Handlungsstränge)
- Die Interrupts sind asynchron, das heißt Zeitpunkt und Häufigkeit ihrer Auslösung sind nicht vorhersehbar

Betrachten wir dazu folgendes Beispiel, in dem eine Industrieanlage mit zwei Temperatursensoren überwacht wird. Beide Temperaturen müssen gleich sein, eine Abweichung signalisiert einen Fehlerzustand der Anlage.[45]

Das Auslesen der Temperatursensoren erfolgt periodisch in einem Timer-Interrupt. Die Ergebnisse werden auf globale Variable geschrieben und im Hauptprogramm ausgewertet. Wir nehmen perfekte Sensoren an, die ohne Rauschen den tatsächlichen Wert liefern.

```
// Beispielcode 1
int iTemp[2];

void interrupt LeseTemperaturen(void) {
// Routine wird durch den Timer-Interrupt periodisch aufgerufen
// ... Vorbereitung
iTemp[0] = <Hardwarezugriff>
iTemp[1] = <Hardwarezugriff>
}

int main(void) {
int iWert0, iWert1;

while(1) {
  iWert0 = iTemp[0];
  iWert1 = iTemp[1];
  if (iWert0 != iWert1) {
    Alarmsirene(LAUT);
  }
}
```

Was ist das Problem mit diesem Code? Der Alarm wird manchmal grundlos ausgelöst! Stellen wir uns vor, die Temperatur in der Anlage steigt langsam an (hochfahren). Dann könnten beide Temperaturwerte "'73"' sein. Das Hauptprogramm weist nun mit

```
  iWert0 = iTemp[0];
```

den Wert "'73"' auf die Variable iWert0 zu. Nehmen wir an, dass genau jetzt durch den Timer der Interrupt ausgelöst wird und auf beide Variablen des Arrays der Wert "'74"' zugewiesen wird. Dann wird mit dem nachfolgenden Befehl

```
  iWert1 = iTemp[1];
```

auf iTemp1 eine "'74"' zugewiesen und der Vergleich ergibt Ungleichheit, Alarm wird ausgelöst.

Der Fehler, der hier auftritt, ist der so genannte *Shared Data Bug*. Er kommt zu Stande, weil der Inhalt von Variablen in einer Programmphase geändert wird, in der nur noch Transport und Verwertung dieses Inhalts stattfinden soll. Eine andere Sichtweise ist, dass das Ergebnis des Programms vom zeitlichen Verhalten einzelner Operationen abhängt, man spricht deshalb auch von *Race Conditions*.

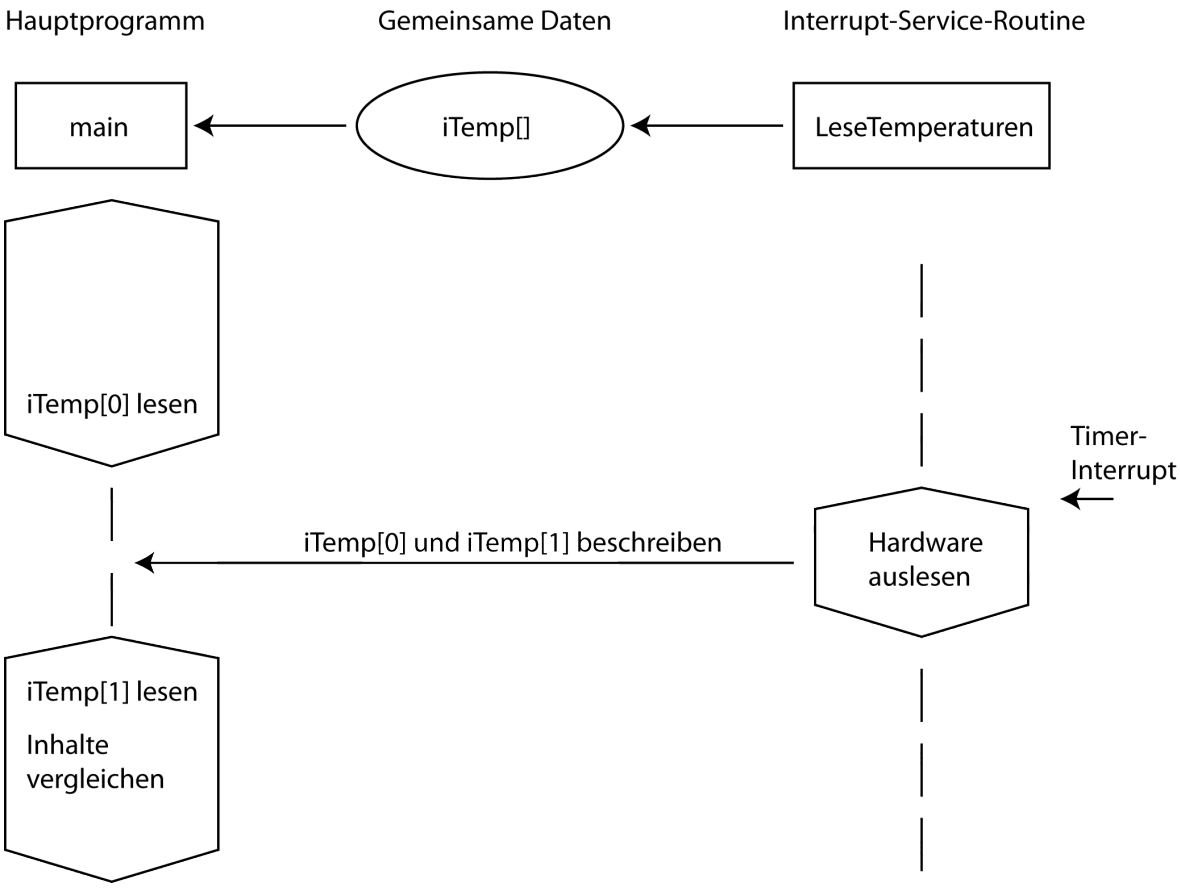


Figure 15.1: Block- und Ablaufdiagramm zum Beispielcode1.

Wir versuchen nun durch eine kleine Veränderung im Code den Shared Data Bug zu beheben. (Beispielcode 2)

```
// Beispielcode 2
int iTemperatur[2];

void interrupt LeseTemperaturen(void) {
// Routine wird durch den Timer-Interrupt periodisch aufgerufen
// ... Vorbereitung
iTemperatur[0] = <Hardwarezugriff>
iTemperatur[1] = <Hardwarezugriff>
}

int main(void) {
while(1) {
  if (iTemperatur[0] != iTemperatur[1]) {
    Alarmsirene(LAUT);
  }
}
```

Scheinbar ist nun das Problem behoben, da die beiden Zugriffe auf die gemein-

samen Daten nun in einem Befehl stehen. Leider besteht aber das Problem immer noch! Der "'if"'-Befehl wird nämlich auf Assembler-Ebene in mehrere Befehle umgesetzt und zwischen diesen kann (und wird) der Interrupt erfolgen. Der "'if"'-Befehl ist daher durchaus unterbrechbar, man sagt auch: Er ist nicht *atomar*.

15.2 Ein weiteres Beispiel

```
// Beispielcode 3
int Sekunden, Minuten, Stunden;

void interrupt Update_Zeit(void) {
// Aufruf der Routine durch den Timer-Interrupt einmal pro Sekunde
// ... Vorbereitung
Sekunde++;
if (Sekunde>=60) {
  Sekunde = 0;
  Minute++;
  if (Minute>=60) {
    Minute = 0;
    Stunde++;
    }
  }
}

int main(void) {
printf("Uhrzeit %2d:%2d:%2d", Stunde, Minute, Sekunde);
}
```

Nehmen wir an, Stunde, Minute und Sekunde haben die Werte 3,59,59. Nun beginnt die Ausgabe mit "'3"', danach kommt der Timerinterrupt und ändert die Werte auf 4,0,0. Ausgegeben wird die Uhrzeit "'3:00:00"', also fast eine Stunde falsch. Der Code für die Zeitausgabe ist nicht atomar, weil die Uhrzeit auf einer mehrteiligen Datenstruktur gespeichert ist und jedes Beschreiben mehrere Maschinenbefehle umfasst.

Ein ähnliches Problem hatte man auf dem ersten PC-Prozessor, dem Intel 8086. Dort wird jede Adresse durch ein Paar von 16-Bit- Registern dargestellt z.B. ist der Befehlszeiger CS:IP (Codesegment-Register: Instruction-Pointer). Um den Befehlszeiger umzusetzen braucht man zwei Befehle: 1. Segmentregister umsetzen, 2. Instruction-Pointer umsetzen. Dort hat man in die Hardware folgende Automatik eingebaut: Wenn ein Zugriff auf das Segmentregister CS erfolgt, werden alle Interrupts für einen Takt gesperrt. In diesem Takt kann dann ungestört den zweiten Teil der Operation vornehmen, nämlich auch das IP-Register zu beschreiben.

Der hier angegebene Code leidet zudem noch an einer weiteren Unsicherheit: Es gibt keine Garantie dafür, in welcher Reihenfolge die Parameter ausgewertet werden.

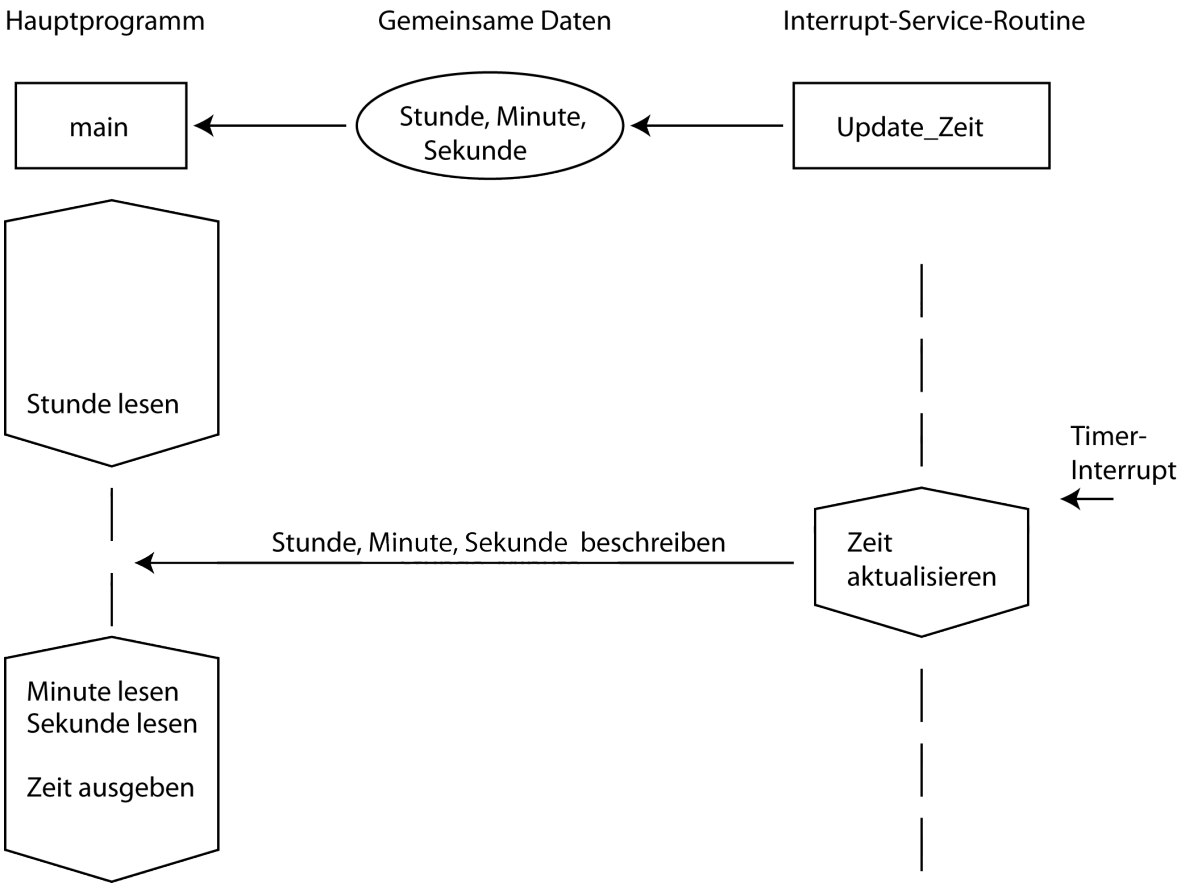


Figure 15.2: Block- und Ablaufdiagramm zum Beispielcode3.

15.3 Nur scheinbar atomar: Code mit verdeckter Mehrteiligkeit

Nehmen wir als Beispiel folgenden Code:

```
// Beispielcode 4
void main(void)
{
  long i,j=15;

  i=j;    // atomar ?
  j=0;

  if (i==j) P1OUT=0xFE;
}
```

Dieser Code wird auf einem 16-Bit-System (MSP 430) auf folgenden Assemblercode umgesetzt:

```
// Beispielcode 4 - Assemblercode

?cstart_end:
main:
 00020C   403C 000F         mov.w   #0xF,R12
 000210   430D              clr.w   R13
   i=j;    // atomar ?
 000212   4C0E              mov.w   R12,R14
 000214   4D0F              mov.w   R13,R15
   j=0;
 000216   430C              clr.w   R12
 000218   430D              clr.w   R13
   if (i==j) P1OUT=0xFE;
 00021A   9C0E              cmp.w   R12,R14
 00021C   2005              jne     0x228
 00021E   9D0F              cmp.w   R13,R15
 000220   2003              jne     0x228
   P1OUT=0xFE;
 000222   40F2 00FE 0021    mov.b   #0xFE,&P1OUT
 000228
}
```

Man sieht, dass die Variablen vom Typ long hier 32 Bit haben und deshalb auf zwei Worte gespeichert werden müssen. Jeder Zugriff auf eine solche Variable (Zuweisung, Vergleich, ...) umfasst daher mehrere Befehle. zwischen diesen Befehlen kann ein Interrupt erfolgen. Der Code ist also nicht atomar muss gegen einen Shared Data Bug geschützt werden.

15.4 Eigenschaften des Shared Data Bug

Der Shared Data Bug entsteht nur, wenn im ungünstigen Moment eine Interrupt-Service-Routine gemeinsame Daten ändert, im letzten Beispiel wäre das zwischen den Assemblerbefehlen, aus denen der "'if"'-Befehl zusammengesetzt ist, dem so genannten *kritischen Codeabschnitt* (critical sections, critical regions). Wenn der gesamte Code groß ist und der kritische Codeabschnitt klein, dann ist die Wahrscheinlichkeit des Shared Data Bug gering. Der Bug tritt also selten auf. Das macht die Sache aber nicht besser, sondern schlimmer, weil man ihn beim Testen schwer findet. Irgendwann tritt er dann aber doch auf, typische Situationen sind:[45]

- Am Freitag Nachmittag um 5 Uhr.
- Wenn der Hardware-Tracer gerade abgebaut ist
- Wenn der Projektleiter/Professor hinter Ihnen steht
- An dem Tag, an dem Sie in Urlaub fliegen wollen und Ihrem Kollegen gerade erklärt haben, dass Ihr Programm fehlerfrei läuft.
- Nachdem Ihr eingebettetes System gerade auf dem Mars gelandet ist
- Und natürlich, bei Kundenvorführungen

Deshalb sollte uns diese Gefahr sehr bewusst sein und wir sollten durch äußerst sorgfältige Programmierung einen Shared Data Bug vorsorglich umschiffen!

15.5 Lösung des Shared Data Problems

Die übliche Lösung besteht darin, dass man die kritischen Abschnitte schützt, indem man während dieser Zeit keine Interrupts zulässt. Dazu wird mit dem Entwicklungssystem immer ein Paar von Funktionen angeboten um Interrupts ein- und auszuschalten. Der Code könnte dann so aussehen.

```
// Beispielcode 4
int iTemp[2];

void interrupt LeseTemperaturen(void) {
// Routine wird durch den Timer-Interrupt periodisch aufgerufen
// ... Vorbereitung
iTemp[0] = <Hardwarezugriff>
iTemp[1] = <Hardwarezugriff>
}

int main(void) {
int iWert0, iWert1;

while(1) {
  disable_ints();
  iWert0 = iTemp[0];
```

```
  iWert1 = iTemp[1];
  enable_ints();

  if (iWert0 != iWert1) {
    Alarmsirene(LAUT);
  }
}
```

Aber auch dieser Code hat seine Schattenseiten. Frage: Welche?

Die Identifikation des kritischen Codes muss der Programmierer selbst übernehmen, es ist bis jetzt kein Tool bekannt, das diese Aufgabe zufrieden stellend löst.

Alle hier besprochenen Probleme (Shared Data Problems) kann man auch als Synchronisationsproblem beschreiben. Der Zugriff auf die gemeinsamen Variablen ist asynchron (zufallsgesteuert) und deshalb ist die Reihenfolge in der die Befehle des Programms ausgeführt werden unbestimmt. Anders ausgedrückt: es gibt mehrere Varianten für diese Reihenfolge.

Das zeitweilige Sperren der Interrupts brachte die Befehle wieder in die beabsichtigte Reihenfolge, stellte also die Synchronisation wieder her. Echtzeitbetriebssysteme bieten elegantere Hilfsmittel zur Synchronisation an: *Semaphore*.

PRAKTIKUMSAUFGABEN NACH DIESEM ABSCHNITT

1. *Aufgabe 16 Test und Analyse eines Interrupt-gesteuerten Programms I*
2. *Aufgabe 17 Test und Analyse eines Interrupt-gesteuerten Programms II*
3. *Aufgabe 18 Test und Analyse eines Interrupt-gesteuerten Programms III*
4. *Aufgabe 19 Empfang einer IR-Impulsfolge und Auslösung eines Interrupts 35*

16 Software-Entwicklung für Mikroprozessoren (Teil V)

16.1 Modellierung mit deterministischen endlichen Automaten

Symbolik bei endlichen Automaten

Ein deterministischer endlicher Automat (englisch Finite State Machine, FSM) ist eine gute Möglichkeit, Software im technischen Bereich zu beschreiben. FSM s sind in der Beschreibungssprache UMl vorgesehen. Die Modellierung mit einer FSM verbessert die Übersicht und bietet einen relativ geradlinigen Weg zu einem funktionierenden Programm. Die FSM-Modellierung ist nicht eindeutig, es gibt oft mehrere Möglichkeiten um einen Programmablauf im Zustandsdiagramm darzustellen.

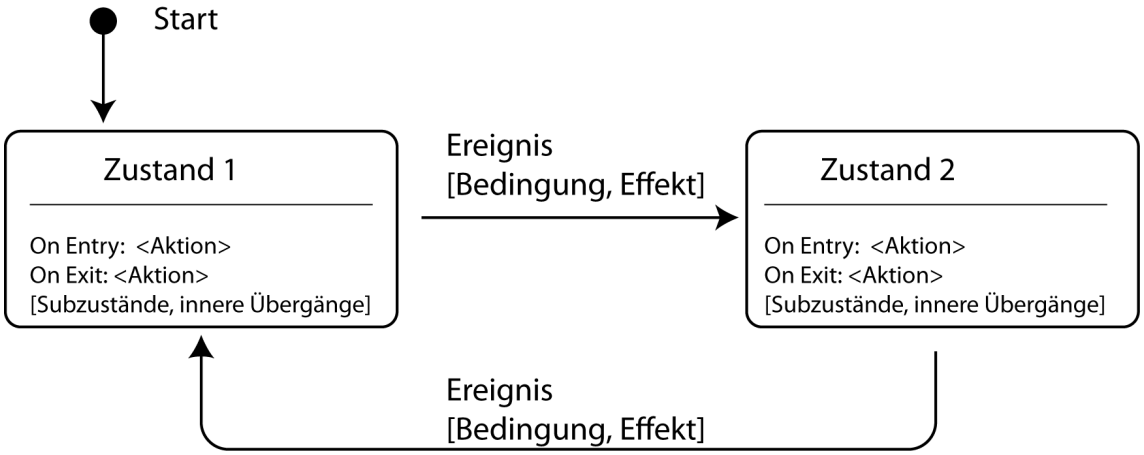


Figure 16.1: Aufbau eines endlichen Automaten.

Rechtecke Zustände

Subzustände Optional: Zusätzliche Unterscheidungen in einem Zustand, zum Beispiel ein Zähler der verschiedene Werte annimmt

On Entry Optional: Aktion, die bei Betreten eines Zustandes ausgeführt wird

On Exit Optional: Aktion, die bei Verlassen eines Zustandes ausgeführt wird

Pfeile Übergänge zwischen Zustände

Ereignis Ereignis das zum Zustandsübergang führt (Z.B. Taste, Pegelwechsel, Timerablauf o.ä.)

Bedingung Optional: Bedingung, die erfüllt sein muss, damit der Zustandsübergang erfolgt

Effekt Optional: Aktion, die beim Zustandsübergang ausgeführt wird

Es ist auch möglich eine fortlaufende Aktion zu definieren, die ständig ausgeführt wird, so lange dieser Zustand besteht (do ...). Beim Entwurf eines endlichen Automaten ist es wichtig, eine klare Vorstellung zu haben und nicht Zustände und Aktionen zu verwechseln. Die Zustände sollten aussagekräftige Namen haben.

Ein endlicher Automat kann leicht in eine Liste umgewandelt werden und aus der Liste kann leicht ein Programm erzeugt werden. So hat man einen geraden Weg, um aus einer Aufgabenstellung zu einem funktionierenden Programm zu kommen. Schöne Beispiele findet man zum Beispiel unter:

FSM am Beispiel eines Toasters

FSM am Beispiel einer Ampelkrezung

FSM ebenfalls am Beispiel einer Ampelkrezung

Beispiel Wechsellicht

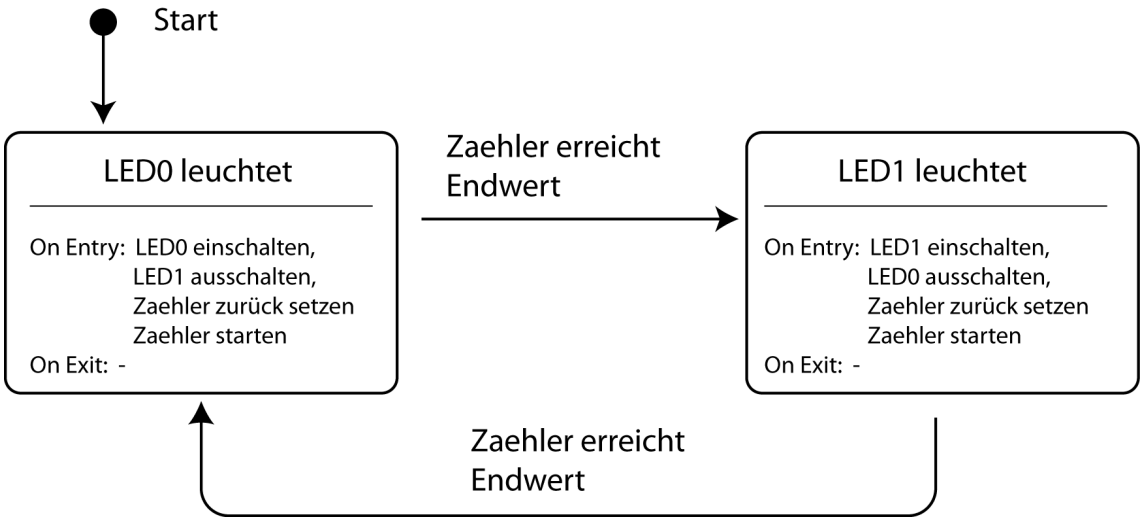


Figure 16.2: Beschreibung eines Wechsellichtes auf LED0 und LED1 mit einem endlichen Automaten. Beim Start leuchtet zunächst LED0.

Diese Grafik kann in folgende Tabelle umgesetzt werden:

Zustand	Beschreibung	Zustandsübergang ausgelöst durch	Folgezustand
Startzustand=1			
1	LED 0 leuchtet, LED1 dunkel	Zähler erreicht Endwert	2
2	LED 1 leuchtet, LED0 dunkel	Zähler erreicht Endwert	1

Bei der Umsetzung in ein C-Programm geht man nach folgendem Muster vor:

- Der Zustand wird durch eine ganzzahlige Variable dargestellt, z.B. "'state"'.
- Der Zustandsautomat wird in einer eigene Funktion realisiert, z.B. "'statemachine"'
- Diese Funktion enthält eine switch-case-Konstruktion, diese enthält die Zustände, Aktionen und Übergänge
- Warteschleifen sind im Zustandsautomaten verboten, sie werden außerhalb des Zustandsautomaten realisiert

Diese Tabelle kann in folgendes C-Programm umgesetzt werden:

```
#define LED0      1
#define LED1      2

unsigned char state = 1;  // Startzustand ist 1

void stateMachine()
{
  switch( state ) {
    case 1:
      Schalte_LEDs(0,1);   // LED0 leuchtet, LED 1 ist dunkel
      state = 2;
      break;

    case 2:
      Schalte_LEDs(1,0); // LED 1 leuchtet, LED 0 ist dunkel
      state = 1;
      break;
  }
}

int main()
{
  ....

  while( 1 ) {
    stateMachine(); // Aufruf des Zustandsautomaten
    Warteschleife( 1000 ); // Wartezeit
  }
}
```

In einer besseren Implementierung kann die Wartezeit durch einen Timer ersetzt werden und die Funktion statmachine wird zur Interrupt Service Routine.

Zum Üben: Schreiben Sie eine saubere Funktion `Schalte_LEDs(1,0)`, die keine Seiteneffekte auf andere Bits von P1OUT hat!

Beispiel Teatimer

Der Tea Timer ("'Küchenuhr"') ist ein abwärts laufender Zähler, der ständig die verbleibenden Sekunden anzeigt und piept, wenn er die Null erreicht hat. Er kann z.B. so modelliert werden:

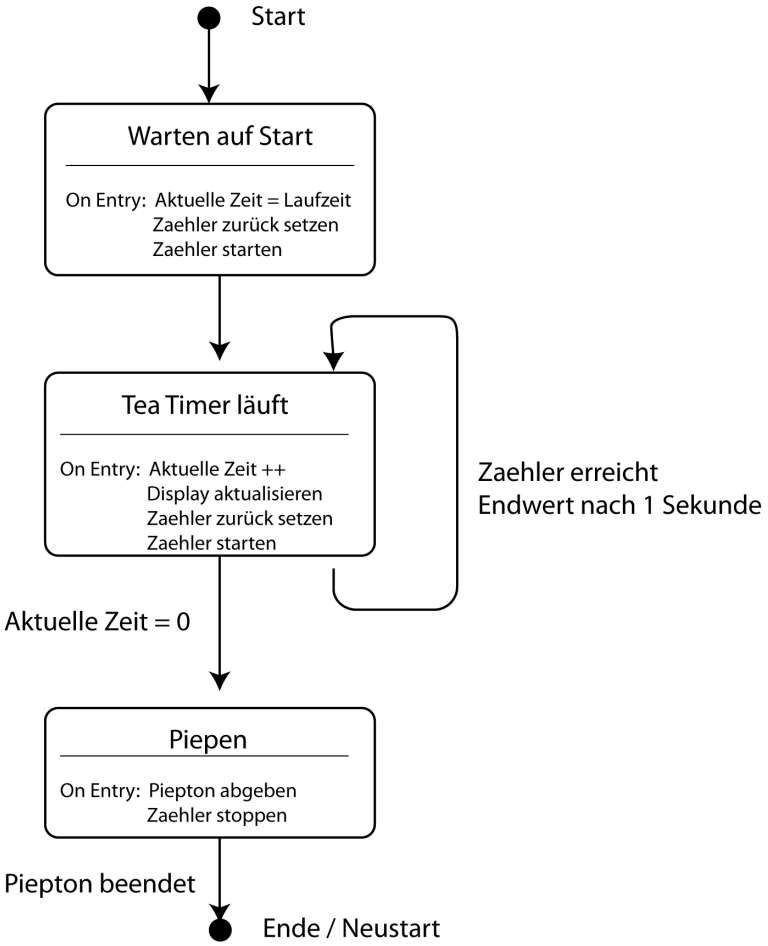


Figure 16.3: Beschreibung eines Teatimers mit einem endlichen Automaten.

PRAKTIKUMSAUFGABEN NACH DIESEM ABSCHNITT

1. *Aufgabe "'Impulsfolge im RC5-Telegramm analysieren und Tastencode ermitteln"'*
2. *Aufgabe "'Steuerung des Tea-Timers mit der Philips Fernbedienung"'*

17 Kommunikations-Schnittstellen

Wichtig für Austausch von Daten mit Mikrocontrollern und anderen Bausteinen wie Sensoren, Displays, Tastaturen, Kartenleser etc.

- Ethernet oder USB sind bei Mikrocontrollern immer noch die Ausnahme.
- Der Datenaustausch wird fast immer seriell aufgebaut um mit wenigen Leitungen auszukommen.
- Es sind asynchrone und synchrone Schnittstellen verbreitet, manche Mikrocontroller haben auch mehrere Schnittstellen.
- Neuere Mikrocontroller haben universelle Schnittstellen, die in mehreren Modi betrieben werden können
- Diese heißen z. B. Universal Serial Communication Interface (USCI) oder Universal Synchronous/Asynchronous Receiver/Transmitter (USART) oder Asynchronous/Synchronous Channel(ASC)

17.1 Asynchrones serielles Interface

Wenn mit Pegeln von +10V / -10V betrieben: RS232

- Datenframes bestehen aus aus einem Startbit, 5 bis 8 Datenbit, einem optionalen Paritätsbit und 1 bis 2 Stoppbits besteht.
- Startbit startet den internen, dieser erzeugt dann Abtastungzeitpunkte für restliche Bits des Datenrahmens
- Keine Taktleitung
- Baudratengenerator ist programmierbar, typisch sind Werte zwischen 1200 Baud und 115200 Baud
- Takt wird durch Teilung eines Mastertaktes erzeugt, Mastertakt ist (nach Vorteilung) 115200 1/s.
- Stoppbit(s) (=Ruhepegel) grenzen den Datenrahmen gegen den nächst folgenden Datenrahmen ab

Wenn eine Pegelanpassung auf RS232 vorhanden ist, kann man problemlos Daten mit einem PC austauschen. Das ist sehr beliebt um Debug-Informationen oder daten an den PC zu geben .

Mikrocontroller – Serielles Interface – Pegelanpassung – USB-Konverter – USB des PCs

17.2 Inter Integrated Circuit Bus, I2C-Bus

Meist kurz als I^2C-Bus oder I2C-Bus bezeichnet

- entwickelt von Fa. Philips für die Kommunikation von ICs untereinander, z.B. zwischen digitalen Bausteinen auf der gleichen Leiterplatte
- Kann nur auf wenige Meter Länge ausgedehnt werden
- synchroner Bus (mit Taktleitung)
- Datenübertragungsrate von maximal 100 kbit/s
- Am I^2C-Bus kann eine größere Anzahl Teilnehmer angeschlossen werden
- Jeder Teilnehmer am I^2C-Bus hat eine Adresse
- jeder Datensatz beginnt immer mit der Adresse des Empfängers.
- Die I^2C-Bus-Schnittstellen können wechselnd als Sender (Master) oder Empfänger (Slave) arbeiten
- Es gibt immer nur einen Master
- Viele I2C-Komponenten im Handel: Sensoren, Displays, Tastaturen, Port-Expander u.a.m.

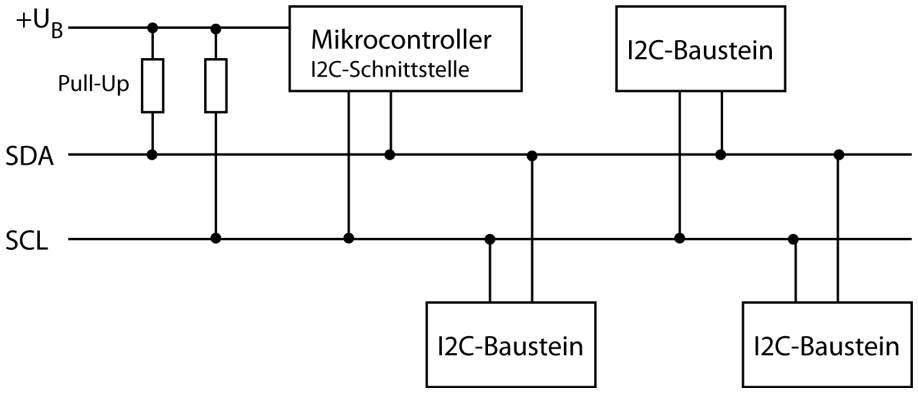


Figure 17.1: Mehrere Bausteine gemeinsam mit einem Mikrocontroller am I^2C-Bus.

Leitungen des I^2C-Bus

Masse Bezugspotenzial für alle Signale

Serial-Data-Line (SDA) Gemeinsame Datenleitung

Serial Clock (SCL) Gemeinsames Taktsignal, signalisiert die Lesezeitpunkte für gültige Datenbits auf der Datenleitung SDA

Anfang und Ende der Übertragung werden durch festgelegte spezielle Signalformen auf SCL und SDA gemeinsam synchronisiert.

17.3 Serial Peripheral Interface, SPI-Bus

- Von der Fa. Motorola (heute Freescale) entwickelt

- Ähnlicher Einsatzbereich wie I^2C-Bus
- Kann über einige Meter ausgedehnt werden
- Separate Leitungen für die beiden Übertragungsrich-tungen
- Alle SPI-Komponenten werden über ein Freigabesignal (Slave Select, SS) aktiviert
- Datenraten bis zu 1 Mbit/s
-

Leitungen des SPI-Bus:

MISO (Master In, Slave Out)

MOSI (Master Out, Slave In)

SCK (Serial Clock)

Ähnlich wie beim I^2C-Bus sind zahlreiche Bausteine mit der SPI-Schnittstelle im Handel.

17.4 CAN-Bus

Der CAN-Bus = Controller Area Network.

- Von der Firma Bosch für den Kraftfahrzeugbereich entworfen
- Die Übertragung ist asynchron, es werden Datenübertragungsraten bis zu 1 Mbit/s erreicht
- Meist differentielle Übertragung über die Leitungen CAN-L und CAN-H übertragen (Differenz der Leitungspotentiale wird gemessen)
- Für den CAN-Bus gibt es einen Arbitrierungsprozess, deshalb kann mit mehreren Busmastern gearbeitet werden
- Bei der CAN-Kommunikation werden nicht einzelne Stationen adressiert, sondern es werden priorisierte Nachrichten verschickt. Die Empfänger entscheiden dann anhand von Masken und Identifizierungsbits, ob sie die Nachrichten empfangen und weiterverarbeiten.
- Bezüglich der Verarbeitung und Verwaltung der empfangenen Nachrichten unterscheidet man zwischen BasicCAN und FullCAN.

Arbitrierung (Wer bekommt den Bus?)

- Notwendig, wenn mehrere Stationen gleichzeitig beginnen zu senden
- Jeder Sender prüft, ob die von ihm gesendeten Daten korrekt auf dem Bus erscheinen
- Falls ja, betrachtet er sich als Bus-Master
- Falls nein, geht er in den Empfangsmodus und versucht später, erneut zu senden.

CAN ist sehr verbreitet, wird im Automobil aber allmählich durch den optischen MOST-Bus ersetzt, dessen Bandbreite auch für Multimediadaten ausreicht.

17.5 IrDA

IrDA = Infrared Data Association, Zusammenschluss von ca. 50 Unternehmen für Datenübertragung mit Infrarotlicht. Manche Mikrocontroller besitzen eine IrDA-Schnittstelle für die drahtlose Übertragung von Daten mit Infrarotlicht nach IrDA 1.0 (bis 115,2kbit/s) oder IrDA 1.1 (bis 4 Mbit/s). IrDA arbeitet mit Infrarotlicht von 850 ... 900 nm.

17.6 Fallbeispiel: Das Universal Serial Communication Interface des MSP430

Das USCI (Universal Serial Communication Interface). ist ein sehr vielseitiger Baustein mit mehreren Modulen. [1]Achtung bei der Internet-Recherche, es gibt auch ein älteres Kommunikations-Interface das USI, funktioniert anders!) Das USCI bietet Hardwareunterstützung für folgende serielle Schnittstellen:

- UART
- IrDA
- SPI
- I^2C

Nicht jeder USCI hat jedes Modul, oft gibt es einen USCI_A und einen USCI_B, Ausstattung oft verschieden.

Im *UART-Mode* (Universal Asynchronous Receive Transmit) kann das USCI Signale erzeugen, die direkt zu einer seriellen Schnittstelle gemäß RS232 kompatibel sind; allerdings nicht mit RS232-Pegel, dazu wird ein externer Pegelwandler gebraucht.

Beim USCI können verschiedene Takte zugeführt werden: Der ACLK-Takt von 32768 Hz oder die SMCLK im MHz-Bereich.

Hat man den hohen Takt anliegen, so kann man im so genannten Oversampling arbeiten: Für jedes Datenbit werden 16 Abtastzeitpunkte erzeugt und mehrfach abgetastet; danach liefert eine Mehrheitsentscheidung den Wert des Datenbits. Das Oversampling-Verfahren ergibt eine größere Robustheit gegen Störungen.

Im einem der Low Power Modes des MSP430 steht evtl. dieser hohe Takt nicht zur Verfügung, sondern nur die 32768 Hz des ACLK-Taktes. Wenn man nun versucht, 9600 Baud durch Herunterteilen von 32768 Hz zu erzeugen, stellt man fest, dass der Teiler 3.41 ist. Man müsste also einen gebrochenen Teiler haben. Lösung: Die so genannte *Modulation*.

- In diesem Zahlenbeispiel wird die Datenleitung manchmal nach drei Takten und manchmal erst nach vier Takten abgetastet.

[1](

- Der Zusatztakt (das Modulationsbit) wird so oft eingeschoben, dass im Mittel die Baudrate von 9600 Baud ungefähr eingehalten wird.
- Datenbits werden nicht immer exakt in der Mitte abgetastet, aber ausreichend nah an der Mitte.(Abb. 17.2)

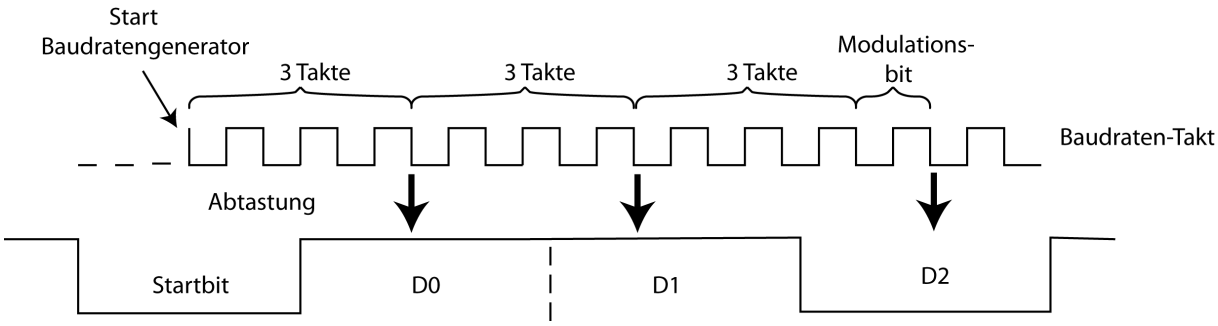


Figure 17.2: Einschieben von Modulationsbits, um Baudraten zu realisieren, die sich nicht durch glatte Teilung aus dem verfügbaren Takt erzeugen lassen.

Programmierung: Im vorliegenden Beispiel wird eine 3 ins Register UCBR (Hauptteiler) eingetragen und 3 ins Register UCBRS (Modulation). Das bedeutet: Abtastzeitpunkt immer nach drei Takten erzeugen und in 8 Takten 3 Modulationsbit zusätzlich eingeschieben. Rechnerischer Teiler: 3 + 3/8 = 3.375, ausreichende Annäherung an 3.41.

Weitere Besonderheiten im UART-Mode sind die automatische Baudratenerkennung und das Encodieren und Dekodieren von IrDA-Signalen (Infrared Data Association).

Im *SPI-Mode* ist ein Betrieb mit 3 oder 4 Leitungen möglich, Frames mit 7 oder 8 Bit und eine einstellbare Taktfrequenz im Master Mode. Im Slave Mode kann die Schnittstelle ohne internes Taktsignal arbeiten, also auch in Low Power Mode 4.

Im *I^2C-Mode* unterstützt das USCI eine Zweidraht-Kommunikation mit anderen I^2C-Bausteinen. Diese kann entsprechend der Spezifikation mit 7- oder 10-Bit-Adressen erfolgen, unterstützt Multimaster-Betrieb, 100 und 400 kBit/s und arbeitet sehr gut auch in den Low Power Modes: Ein Auto-Wake-Up ist möglich und auch ein Slave-Betrieb in LPM4.

PRAKTIKUMSAUFGABEN NACH DIESEM ABSCHNITT

1. *Aufgabe 22 Datenübertragung zur RS232-Schnittstelle des PCs*

18 Systembus und Adressverwaltung

18.1 Busaufbau

Was ist ein Bus?

In einem Rechnersystem finden pausenlos Datenübertragungen über elektrische Leitungen statt. Mehrere Leitungen parallel ergeben eine höhere Datenrate. Wenn man von jedem Baustein zu jedem anderen ein Bündel Leitungen zieht ergeben sich sehr viele Leitungen und viele Kreuzungen auf der Platine, die Platine wird teuer. Außerdem lässt sich dann kein weiterer Baustein einbauen, weil die Leitungen fehlen. Daher: Ein *Bus*. (lat. omnibus = alle)

Ein Bus ist ein Bündel parallel geführter Leitungen die eine gemeinsame Ansteuerung haben und an die mehrere Bausteine parrallel angeschlossen sind.

Vorteile:

- Insgesamt viel weniger Leitungen,
- Der Bus ist erweiterbar für neue Bausteine.

Man unterscheidet die Busleitungen nach der Art der übertragenen Daten in *Datenbus*, *Adressbus* und *Steuerbus*. Ein typisches Beispiel: Der Bus eines Mikroprozessorsystems mit 32 Datenleitungen, 32 Adressleitungen und 29 Steuerleitungen. Die 32 Datenleitungen sind alle gleich beschaltet und übertragen bidirektional die 32 Datenbits D_0 – D_{31}. Der Adressbus ist unidirektional und überträgt die 32 Adressbits A_0 – A_{31}. Die Leitungen des Steuerbusses sind heterogen, jede Leitung hat eine andere Aufgabe. Beispiele für Bussysteme sind PCI, ISA- und EISA-Bus der PCs.

- Computer mit *Harvard-Architektur* haben getrennten Daten- und Programmspeicher (Abb. 18.1),
- Computer mit *von Neumann-Architektur* haben gemeinsamen Daten- und Programmspeicher (Abb. 18.2).

Eine weitere Klasse sind die Feldbusse, die z.B. in der Industrieautomation Geräte verbinden,wie IEC-Bus, SCSI-Bus, Profi-Bus und INTERBUS-S.

Es gibt nicht nur bitparallele sondern auch bitserielle Busse, die mit einer Datenleitungen auskommen. Z.B. verbindet der I^2C-Bus (Inter Intergrated Circuits) integrierte Schaltungen auf Platinen und in Geräten. Der USB (Universal serial Bus) verbindet den PC mit Peripheriegeräten aller Art, der CAN-Bus und der FlexRaybus verbinden Komponenten im Auto.

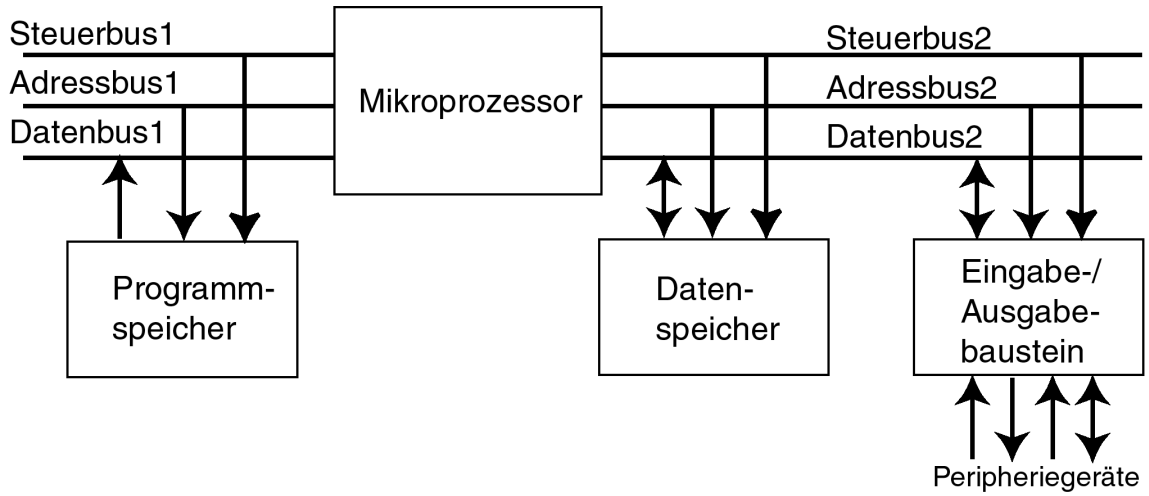


Figure 18.1: Mikroprozessorsystems mit Harvard-Architektur

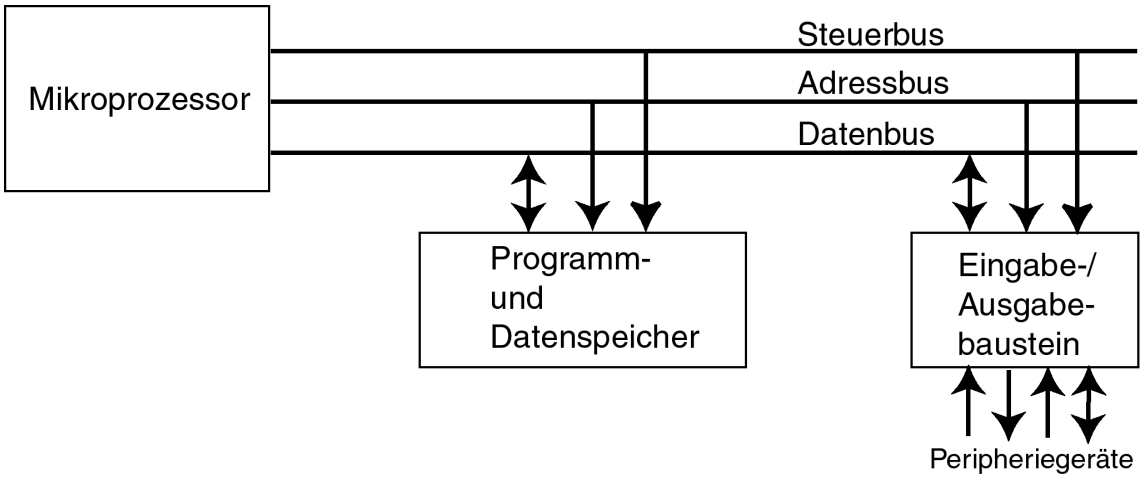


Figure 18.2: Mikroprozessorsystems mit von Neumann-Architektur

- Auf dem Bus fließen die Daten in wechselnde Richtungen
- Am Bus darf immer nur maximal ein Baustein als Ausgang d.h. Sender aktiv sein.
- Nicht aktive Ausgänge dürfen die Busleitungen nicht beeinflussen.
- Es muss für alle Operationen einen streng definierten Ablauf geben, das *Busprotokoll*.
- Der Eingangslastfaktor der Empfängerbausteine darf nicht zu groß und der Ausgangslastfaktor der Senderbausteine nicht zu klein sein.

Busankopplung: Bustreiber mit Tristate-Ausgängen

Ein Tristate-Ausgang kennt drei Zustände: (Tristate = "'3 Zustände"')

- HIGH

- LOW
- hochohmig (abgekoppelt, floating, High-Z, Ausgang passt sich jedem äußeren Potenzial an)

Um die obigen Regeln für den Busbetrieb zu erfüllen braucht man zur Busankopplung spezielle Schaltungen. Dazu wird meistens ein *Bustreiber* (engl. Bus Driver, Buffer) mit Tristate-Ausgang benutzt.

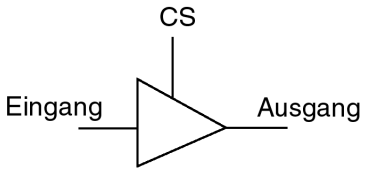


Figure 18.3:
Ein Treiber ist eine Verstärkungsstufe mit einem Eingang, einem Tristate-Ausgang und einem Chip-Select-Steuereingang.

Treiberbausteine verstärken das Signal, so dass an dieser Leitung mehr Bausteine angeschlossen werden können. Tristate-Treiber haben einen Enable-Eingang an dem sie aktiviert werden. Der Ausgang

1. Enable-Signal aktiv: Ausgangssignal ist gleich dem (verstärkten) Eingangssignal.
2. Enable-Signal passiv: Ausgang ist hochohmig, das heißt er ist praktisch abgekoppelt, führt kein Signal und passt sich dem Potenzial der angeschlossenen Busleitung an.

Das ergibt folgende Wahrheitstabelle:

EN (Enable)	Eingang	Ausgang
L	L	hochohmig
L	H	hochohmig
H	L	L
H	H	H

Viele Bausteine müssen manchmal Daten senden und manchmal Daten empfangen. Dazu werden umschaltbare *bidirektionale Bustreiber* (engl. *Bus Transceiver*) benutzt (Abb. 18.4). Bustreiber sind entweder separate Bausteine (z.B. 74LS245) oder in andere ICs integriert.

18.2 Ein- und Ausgabe (E/A)

Engl. Bezeichnungen:
Input/Output, I/O, IO,

Bausteine:
Inputports, Outputports, IO-Ports, kurz Ports, I/O-Kanäle, I/O-Devices

Aufgabe:
Übergabestelle zwischen Systembus und anderen externen oder internen digitalen Bausteinen, (Port=Hafen), Synchronisation, Pegelanpassung

Beispiele:
Tastatur-Controller, Netzwerkschnittstelle, Echtzeituhr, Laufwerkscontroller

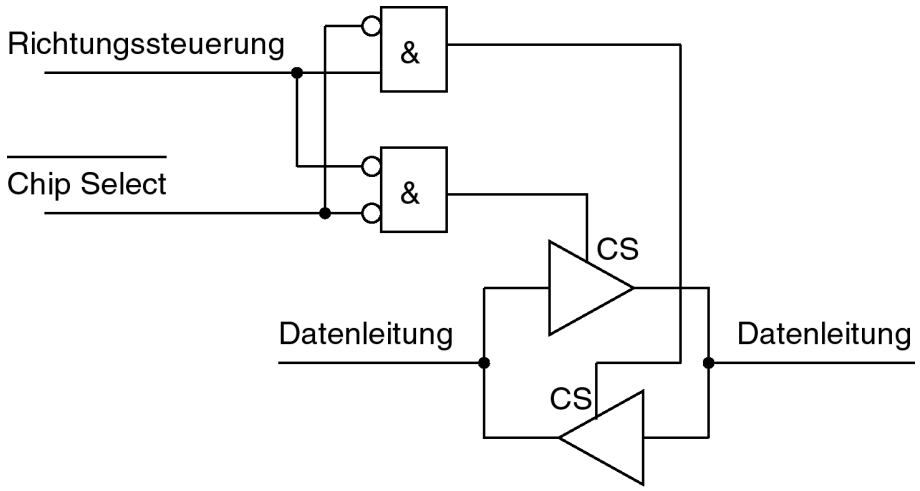


Figure 18.4: Ein bidirektionaler Bustreiber kann wahlweise in beiden Richtungen arbeiten oder die Leitungen trennen. Es ist hier nur eine Datenleitung gezeichnet.

Ankopplungsschaltungen

Die E/A-Schaltungen müssen gewährleisten, dass der Betrieb auf dem Bus nicht gestört wird:

Eingabe: Die externen Signale nur bei Ausführung eines Eingabebefehls kurz auf den Datenbus legen, bis der Prozessor sie übernommen hat.

Ausgabe: Die vom Prozessor über den Bus kommenden Signale bei Ausführung eines Ausführungsbefehls über den Datenbus in einen Zwischenspeicher übernehmen, von dort an die Peripherie geben und bis zur Ausgabe eines neuen Wertes stabil halten.

Einfache Eingabeschaltung: Tristate-Bustreiber

Aufgabe der Eingabeschaltung:

- Die Eingabeschaltung muss die von aussen kommenden TTL-Pegel übernehmen und *erst auf Anforderung des Prozessors* an den Datenbus übergeben.
- Für jede Datenleitung ist ein Treiber mit Tristateausgang und einem gemeinsamen Enable-Eingang vorgesehen.
- Der Tristateausgang ist am Datenbus des Mikroprozessors angeschlossen, der Eingang der Schaltung an der Peripherie.(Abb. 18.5)
- Ein Tristate-Treiber hat drei Zustände an seinem Ausgang: HIGH, LOW und inaktiv
- Ein Tristate-Treiber gibt das Eingangssignal erst an seinen Ausgang weiter, wenn er mit dem Enable-Signal aktiviert wird.

Signale:
IORQ (IO-Request) vom Prozessor

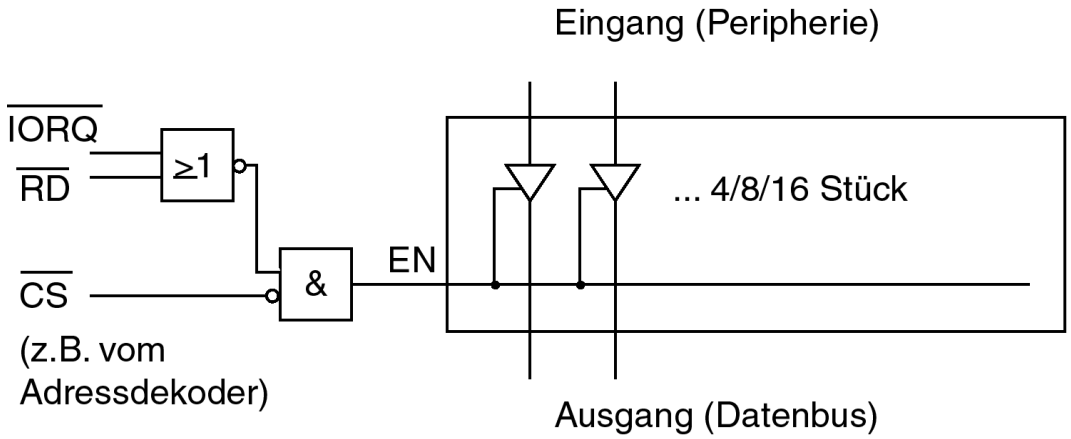


Figure 18.5: Eine Eingabeschaltung besteht aus Treibern mit Tristateausgang und einem gemeinsamen Freigabeeingang EN. Über die Steuerleitungen (hier $\overline{IORQ}, \overline{RD}$ und $\overline{CS}$) schaltet der Prozessor die Treiber frei.

RD (Read) vom Prozessor
CS (Chip Select) von Adressdekodierlogik (siehe später)

Eingabevorgang:

- Die Tristate-Ausgänge der Eingabeschaltung sind zunächst hochohmig
- Für eine Eingabe werden die Tristate-Treiber über Steuerleitungen durch den Prozessor (kurz) freigeschaltet.
- Die von der Peripherie kommenden Signale werden auf den Datenbus durchgeschaltet
- Der Prozessor übernimmt in einem Lesezyklus die Daten vom Bus
- Die Eingabeschaltung wird wieder deaktiviert, die Treiber werden wieder inaktiv.
- Der Bus ist wieder frei für andere Datentransfers
- Die Daten können im Prozessor verarbeitet werden; aus dem Signalmuster (HLLHHHL...) ist nun ein entsprechendes Bitmuster (1001110...) im Prozessor geworden

Einfache Ausgabeschaltung: Flipflops mit Bustreiber

Aufgabe der Ausgabeschaltung:

- Die Ausgabeschaltung muss die vom Prozessor kommenden TTL-Pegel nach außen weitergeben
- Sie muss die Signale halten, auch wenn auf dem Datenbus schon wieder andere Signale (Bitmuster) anliegen
- Dazu müssen die Bitmuster im Ausgabebaustein zwischengespeichert werden; er enthält in jeder Leitung ein Flipflop

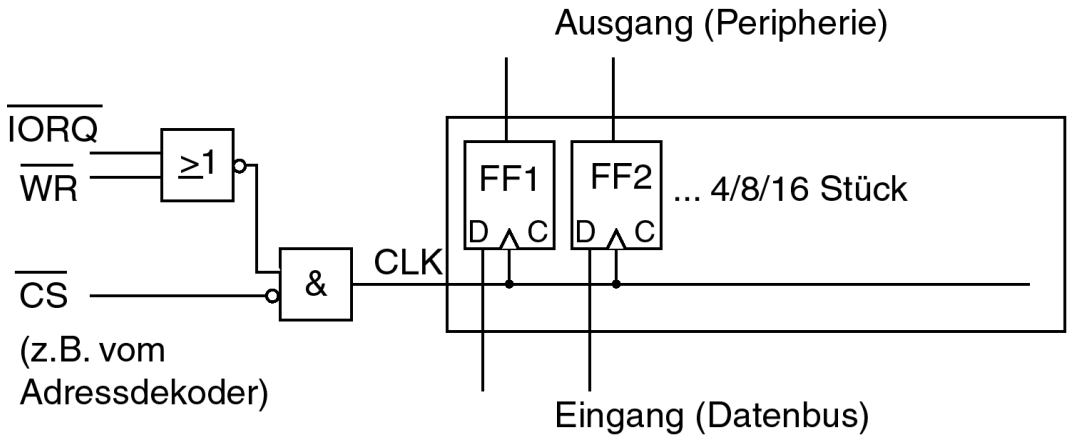


Figure 18.6: Eine Ausgabeschaltung besteht aus Flipflops mit einem gemeinsamen Clock-Eingang CLK. Über die Steuerleitungen (hier $\overline{IORQ}, \overline{WR}$ und $\overline{CS}$) schaltet der Prozessor den Clock-Eingang frei.

Signale:
IORQ (IO-Request)
WR (Write) vom Prozessor
CS (Chip Select) von Adressdekodierlogik (siehe später)

Ausgabevorgang:

- Der Prozessor gibt ein Bitmuster auf den Datenbus aus.
- Die Eingänge der Speicher-Flipflops werden frei geschaltet, jedes FF speichert ein Bit ein.
- Am Ausgang der Speicher-Flipflops kommen die gespeicherten Daten als HIGH oder LOW auf die Peripherieleitungen
- Die Eingänge der Speicher-Flipflops werden wieder gesperrt
- Der Datenbus ist wieder frei für andere Datentransfers
- das aus dem Prozessor kommende Bitmuster (1001110...) ist nun ein Signalmuster (HLLHHHL...) an den Ausgangsleitungen geworden

Geräte-Controller

Komplexe Peripheriegeräte wie z.B. Laufwerke werden durch *Controller* gesteuert. Der Controller allein stellt die Verbindung zum angeschlossenen Gerät her. Er besitzt eine Reihe von Registern (=Gruppe von Speicher-Flipflops mit gemeinsamer Steuerung). Die Register sind durch Ein-/Ausgabeschaltungen an den Bus angekoppelt. Es gibt drei Arten Register:

Datenregister nehmen Daten auf, werden beschrieben und gelesen.

Statusregister enthalten Statusinformetionen über das angeschlossenen Gerät auf, werden vom Prozessor nur gelesen.

Steuerregister steuern die Arbeitsweise des Controllers, und werden vom Prozessor beschrieben.

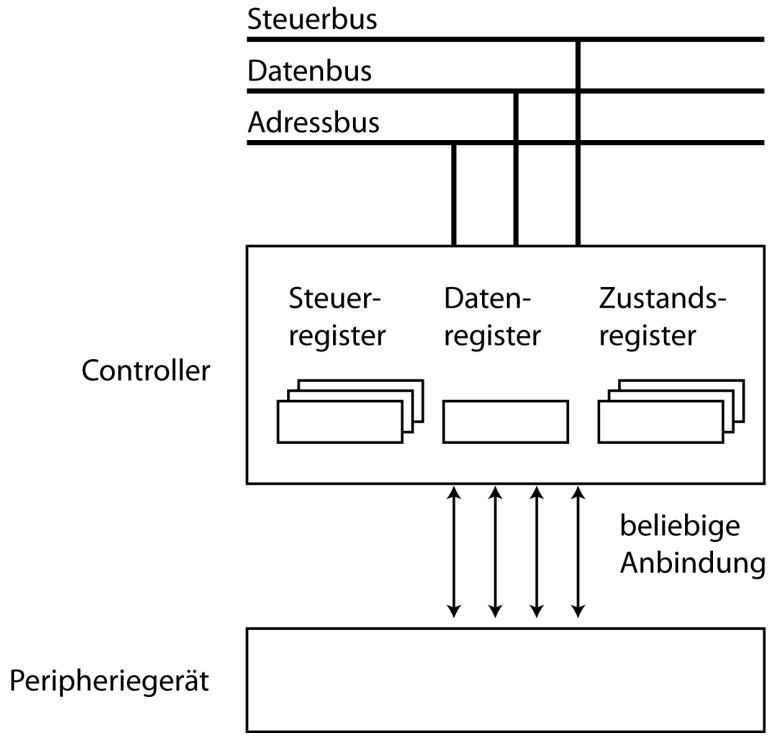
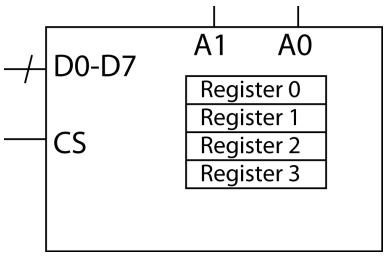


Figure 18.7: Ein Controller stellt die Verbindung zu einem komplexen Peripheriegerät her.

Die Software zum korrekten Ansprechen des Controllers nennt man Gerätetreiber, sie gehört zum Betriebssystem. Die Register des Controller werden ausgewählt, indem man an den Adresseingängen des Controllers die binär codierte Registernummer anlegt. In Abb.18.7 ist ein Controller mit 4 Registern gezeigt, in der Tabelle sieht man die Zuordnung der Adressen zu den Registern.



A_1	A_0	adressierter interner Speicherplatz
0	0	Register 0
0	1	Register 1
1	0	Register 2
1	1	Register 3

Der zeitliche Ablauf eines Lesezugriffs auf einen EA-Baustein (Eingabezyklus) sieht ungefähr so aus:

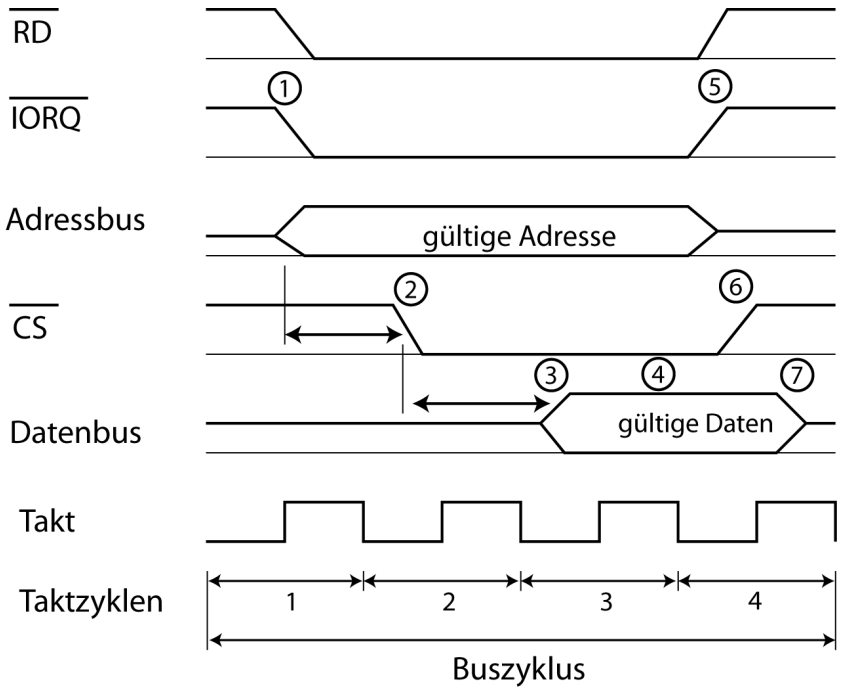


Figure 18.8: Timing eines Eingabezyklus.

Der Buszyklus besteht in diesem Beispiel aus vier Taktzyklen. Ablauf:

Zeitpunkt 1 Prozessor legt RD, IORQ und die Adresse des E/A-Bausteins auf die Busleitungen

Zeitpunkt 2 Adressdekodierlogik hat Adresse erkannt und erzeugt CS für diesen E/A-Baustein

Zeitpunkt 3 E/A-Baustein liefert Daten

Zeitpunkt 4 Datenübernahme durch Prozessor

Zeitpunkt 5 Prozessor nimmt RD, IORQ und Adresse des E/A-Bausteins von den Busleitungen

Zeitpunkt 6 Adressdekodierlogik nimmt CS zurück, da Adresse nicht mehr auf Bus

Zeitpunkt 7 E/A-Baustein schaltet Datenausgänge hochohmig, da kein CS mehr anliegt

Ein programmierbarer E/A-Baustein ist der PPI 8255 (Datenblatt)

18.3 Busanschluss und Adressverwaltung

Allgemeines

Durch die Ausgabe eines Bitmusters auf dem Adressbus wird gezielt eine ganz bestimmte Speicherzelle oder ein bestimmter E/A-Baustein angesprochen. Das ausgegebene Bitmuster nennt man auch die *Adresse* (Abb. 18.9). Durch die gewählte Adresse und die Steuerleitungen wird gezielt ausgewählt:

- ein Baustein am Bus, das kann ein Speicherchip oder ein E/A-Gerät sein.
- innerhalb des Bausteines ein Speicherplatz/Register

ein Speicherplatz in angesprochen.

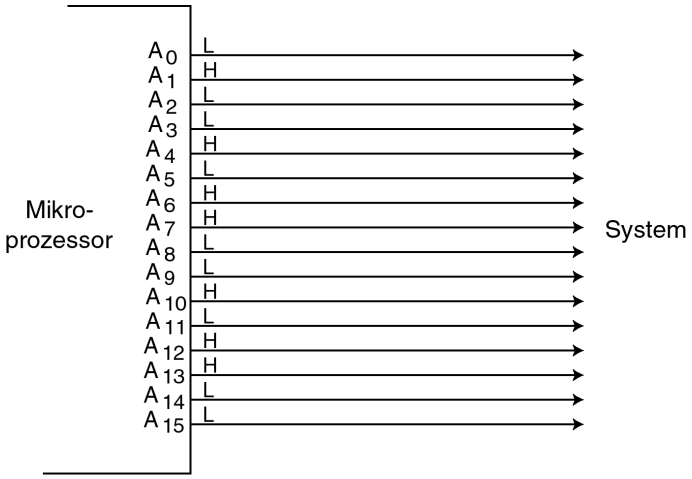


Figure 18.9: Auf einen 16-Bit-Adressbus wird die Adresse 34D2h ausgegeben. Die Beschaltung des Busses bestimmt, welcher Busteilnehmer unter dieser Adresse und welche Speicherzelle/welches Register darin angesprochen wird.

Speicherbausteine haben sehr viele interne Speicherplätze, denn gerade das ist ihr Zweck.

E/A-Bausteine haben dagegen meistens nur wenige interne Speicherplätze, nämlich nur die für den Betrieb notwendigen Status-, Steuer- und Datenregister.

*E*in Baustein mit k nicht gemultiplexten Adresseingängen kann maximal 2^k interne Speicherplätze besitzen.

Beispiele:

PPI 8255, Programmierbarer E/A-Baustein , 4 interne Speicherplätze, 2 Adressleitungen, um die $4 = 2^2$ internen Plätze zu adressieren, nämlich A_0 und A_1.

Ein EEPROM mit einer Organisation von 4kx8Bit , 12 Adressleitungen, um die 4096 = 2^{12} internen Speicherplätze zu adressieren nämlich $A_0 - A_{11}$.

Ein Memory-Controller für eine Speicherbank von 256 MByte Da 256 MByte = 2^{28} Byte ist, 28 Adressleitungen (auch wenn intern mit Adressmultiplexing gearbeitet wird).

- Es muss sichergestellt sein, dass unter jeder Adresse immer nur *ein* Baustein angesprochen wird, gleichgültig ob Speicher- oder E/A-Baustein (Vermeidung von Adress-Konflikten).
- Der Adressraum sollte möglichst gut ausgenutzt werden, d.h. jeder Baustein sollte nur unter einer Adresse zu finden sein (keine *Spiegeladressen*).
- Die Adressräume von Speicherbausteinen müssen lückenlos aufeinander folgen.
- Jeder interne Speicherplatz bzw. jedes Register erscheint unter einer eigenen Adresse im Systemadressraum. Die Systemadressen der internen Speicherplätze und Register eines Bausteines sollen in der Regel zusammenhängend sein.

Adressdekodierung von E/A-Bausteinen

E/A-Bausteine sind dagegen Einzelsysteme und bilden keine miteinander zusammenhängenden Adressbereiche. Zwischen den Adressen verschiedener E/A-Bausteine dürfen also Lücken bleiben. Aber auch die Register *eines* E/A-Bausteines sollten im Systemadressraum einen zusammenhängenden Block bilden.

Um das zu erreichen wird der Adressbus*geteilt*. Die k niedrigstwertigen Adressleitungen werden direkt an die Adresseingänge des Bausteines geführt und dienen zur Auswahl des richtigen internen Speicherplatzes oder Registers. Die nächstfolgenden l Adressleitungen werden zur Adressdekodierung auf einen *Adressdekoder* geführt (Abb. 18.10).

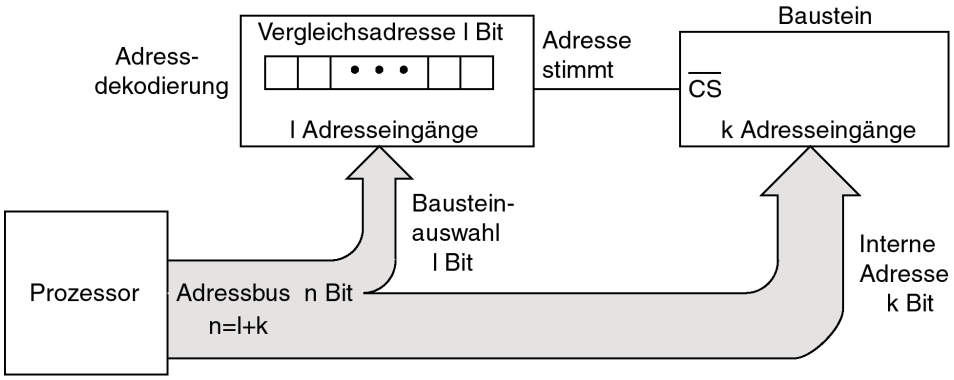


Figure 18.10: Für die korrekte Freischaltung von Speicher- oder Ein-/Ausgabebausteinen wird der Adressbus aufgeteilt.

Der Adressdekoder vergleicht diese l Bits mit einer intern eingestellten Vergleichsadresse (Bausteinadresse) und schaltet den angeschlossenen Baustein frei, wenn die beiden Bitmuster gleich sind. Diese l Leitungen reichen zur Freischaltung von maximal 2^l Bausteinen aus. Wenn ein Adressbus mit n Leitungen benutzt wird und keine Leitung unbenutzt bleibt, gilt

$$k + l = n \tag{18.1}$$

Diese Situation kann übersichtlich mit dem *Adress-Aufteilungswort* (auch kurz

Adresswort) dargestellt werden, einem Schema in dem die Funktion der Adressbits gruppenweise erkennbar wird (Abb. 18.11).

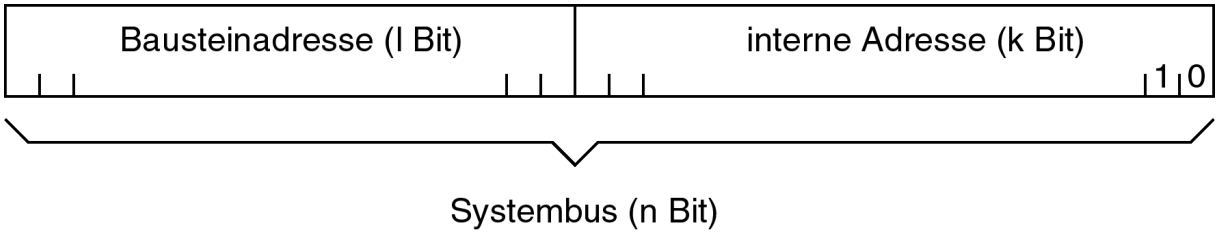


Figure 18.11: Das Adress-Aufteilungswort beim Busanschluss von Bausteinen, wenn alle Adressleitungen benutzt werden.

Als Adressdekoder kann ein digitaler *Vergleicher* (Komparator) benutzt werden. Ein solcher Vergleicher hat zwei Reihen von Digitaleingängen, die z.B. A- und B-Eingänge genannt werden. Der Ausgang des Vergleichers ($\overline{A = B}$) zeigt an, ob die Bitmuster an den beiden Eingangsreihen exakt gleich sind (Abb. 18.12). Ein Beispiel für einen kommerziellen digitalen Komparator ist der 74688 (siehe Datenblatt)

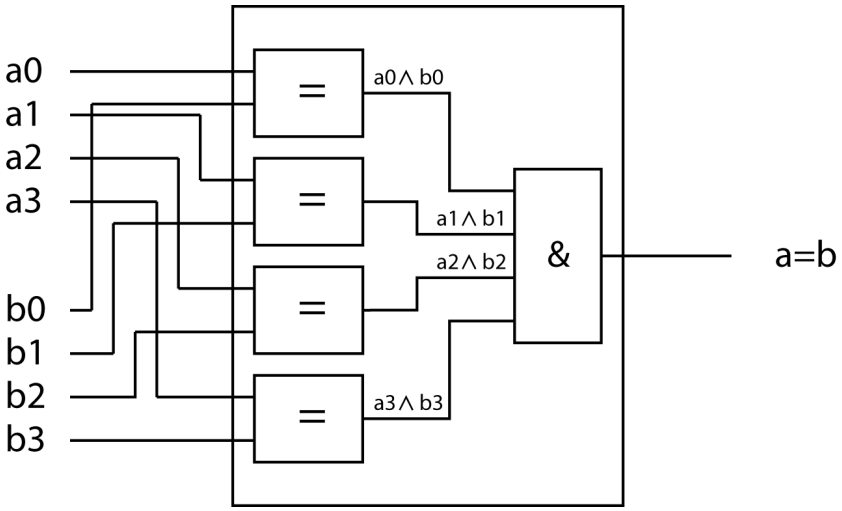


Figure 18.12: Ein 4-Bit-Adressdekoder.

Mit dem Adressdekoder wird für einen einzelnen E/A-Baustein wie folgt eingesetzt:

- An die eine Eingangsreihe des Vergleichers (A-Eingänge) legt man die Adressleitungen zur Bausteinauswahl
- An die andere Eingangsreihe des Vergleichers (B-Eingänge) legt man eine interne, fest eingestellte Vergleichsadresse an.
- der digitale Vergleicher stellt fest ob beide Adressen gleich sind und schaltet dann den betroffenen E/A-Baustein frei.

Das Freischaltungssignal muss u.U. mit weiteren Signalen verknüpft werden, bevor es an den Chip-Select-Eingang des Bausteines geht, z.B. bei isolierter E/A-Adressierung (s. Abschn. 18.3) mit dem $Mem/\overline{IO}$-Signal. Der Busmaster spricht den Baustein

an, indem er in dem Bitfeld zur Bausteinauswahl genau das Bitmuster erscheint, auf das der Adressdekoder eingestellt ist. Dies geschieht durch die richtige Adresse im Befehl. Das fest eingestellte Bitmuster am Adressdekoder bestimmt also die Adressen des Bausteins im Systemadressraum.

Beispiel Ein E/A-Baustein soll mit einem Adressdekoder freigeschaltet werden. Der für das Beispiel ausgewählte E/A-Baustein hat vier interne Register, die man über die beiden Adresseingänge A_0 und A_1 folgendermaßen adressieren kann:

A_1	A_0	Interne Adresse
0	0	Reg 0
0	1	Reg 1
1	0	Reg 2
1	1	Reg 3

Der Baustein sollte auch im Systemadressraum genau 4 Adressen belegen. Der Bus soll insgesamt 10 Adressleitungen haben, davon werden 2 in den Baustein geführt und 8 auf den Adressdekoder. Es ist also in Gl. 18.1: $l = 8;\ k = 2;\ n = 10$. Daraus ergibt sich das Adresswort, das in Abb. 18.13 gezeigt ist. Vollständige Schaltung s. Abb.18.14.

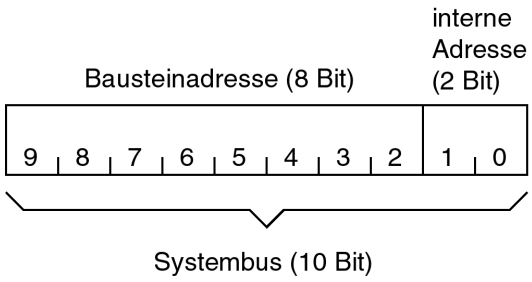


Figure 18.13:
Das Adresswort zu der Schaltung in Abb. 18.14.

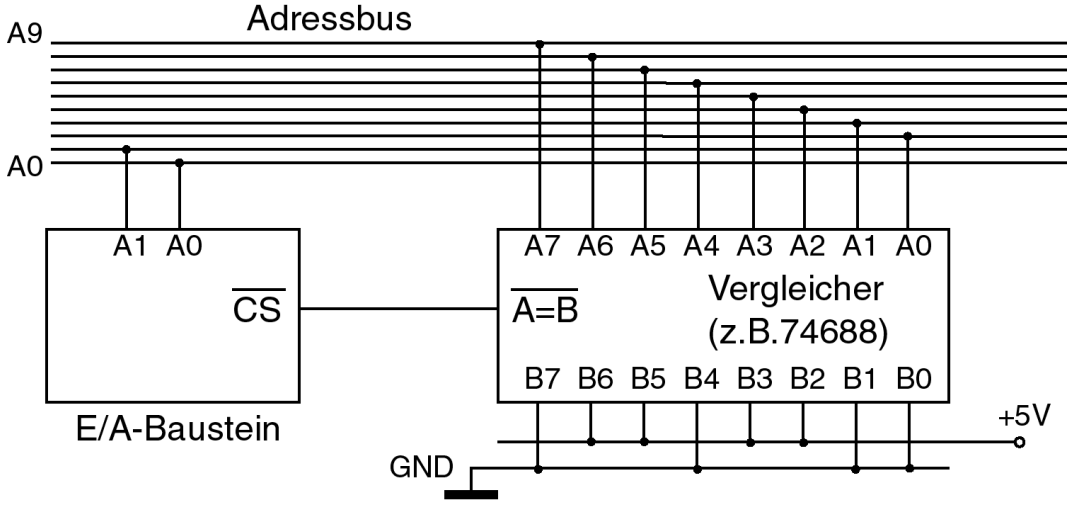


Figure 18.14: Die Adressdekodierung eines E/A-Bausteines mit einem Vergleicher, der die Adressbits A_2 – A_9 prüft.

Um die belegten Adressen zu bestimmen, kann man sich anhand einer Tabelle überlegen, bei welchen Bitmustern der Chip freigeschaltet wird und welche inter-

nen Adressen jeweils erreicht werden. Damit der Baustein über $\overline{CS}$ freigeschaltet wird, müssen die Bitmuster an den Eingängen A und B übereinstimmen. Auf Grund der Beschaltung an den B-Eingängen muss das Signalmuster an den A-Eingängen also LHHLHHLL und damit das Bitmuster auf den Adressleitungen A_9 – A_2 immer gleich 01101100 sein, um den Baustein überhaupt freizuschalten. Das Bitmuster auf den Bits A_0, A_1 bestimmt welche interne Adresse angesprochen wird; dort sind alle Bitmuster erlaubt. Nach diesen Regeln entsteht folgende Tabelle,

Adresse binär	Adresse hex.	Interne Adresse
01101100 00	1B0	Reg 0
01101100 01	1B1	Reg 1
01101100 10	1B2	Reg 2
01101100 11	1B3	Reg 3
sonst	—	—

Unter der Adresse 1B0h spricht man also den Speicherplatz mit der internen Adresse 0 an, diese nennt man auch die *Basisadresse*. Wegen der Verschiebung um 2 Bit ist die Basisadresse ist hier gleich dem vierfachen des Bitmusters auf der B-Seite, wenn es als binäre Zahl aufgefasst wird. Alle anderen Adressen des Bausteines schließen sich nach oben an die Basisadresse an. Der *Adressbereich* des Bausteines im System ist 1B0h – 1B3h. (Abb. 18.15)

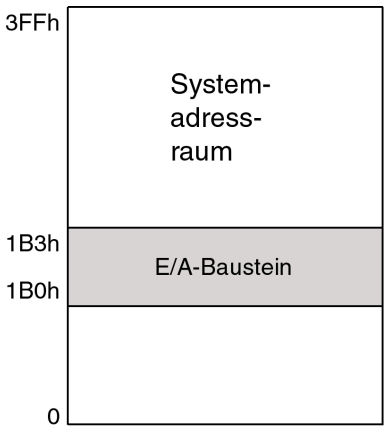
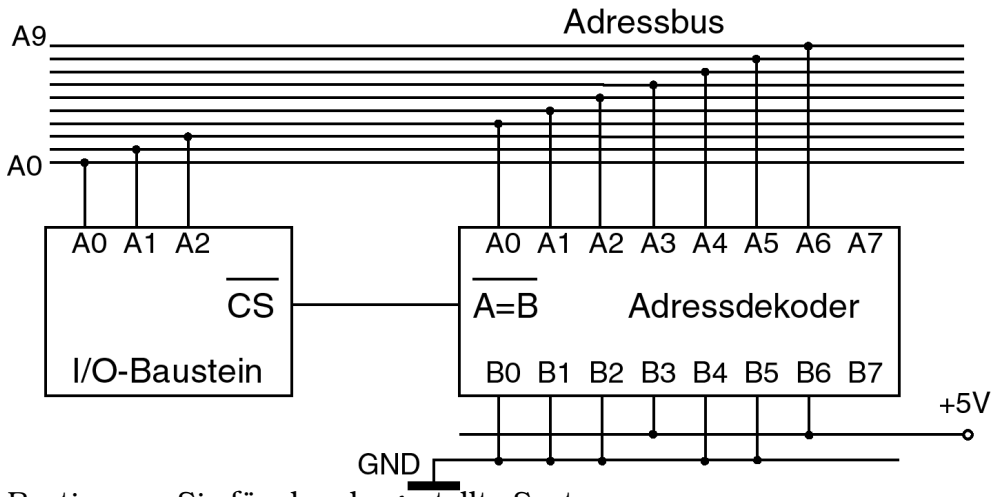


Figure 18.15:
Die vier Adressen des E/A-Bausteines bilden einen zusammenhängenden Bereich im Systemadressraum.

Weiterentwicklung um mehr Flexibilität zu erhalten:

- B-Eingang mit DIP-Schaltern (Dual Inline Package-Miniaturschalter die auf der Platine eingelötet werden)
- Flipflops an den B-Eingängen: Jetzt kann das Betriebssystem beim Bootvorgang die E/A-Adressen in die Flipflops einschreiben: Plug and Play

Übung: 18.1 Adressberechnung



Bestimmen Sie für das dargestellte System
a) welche und wie viele Adressen der I/O-Baustein im System belegt,
b) das Adress-Aufteilungswort,
c) die Basisadresse.

Übung: 18.2 Tristate-technik

Beschreiben Sie die drei Zustände eines Tristate-Ausgangs

Adressdekodierung bei Speicherbausteinen

Hauptunterschied zur Verwaltung von E/A-Adressen:

- Es werden mehrere identische Bausteine eingesetzt
- Die Systemadressen müssen einen lückenlosen Block bilden
- Die Bausteine haben sehr viele interne Speicherplätze

Man benutzt deshalb Dekodierbausteine, die mehrere Speicherbausteine aktivieren können: 1-aus-n-Dekoder Das Bitmuster auf dem Adressbus (Adresse) wird wieder aufgeteilt in verschiedene Felder,

- Baustein-Nummer
- Baustein-interne Adresse

Es können auch größere Einheiten gebildet werden: Mehrere Chips bilden ein Modul (z. B. DIMM, Dual Inline Memory Module) und mehrere Module sind auf einer Karte.

In einem System mit 32-Bit-Datenbus will man natürlich die Daten auch in Paketen von 32 Bit schreiben bzw. lesen. Der Speicher ist aber nach wie vor Byte-adressiert, das heißt unter jeder Speicheradresse ist ein Byte gespeichert. Ein 16-Bit-Datum belegt somit zwei Adressen, ein 32-Bit-Datum vier. Wenn man Speicherbausteine mit 32 Datenleitungen benutzt, kann man das Speichermodul so aufbauen, wie es in Bild 18.16 gezeigt ist.

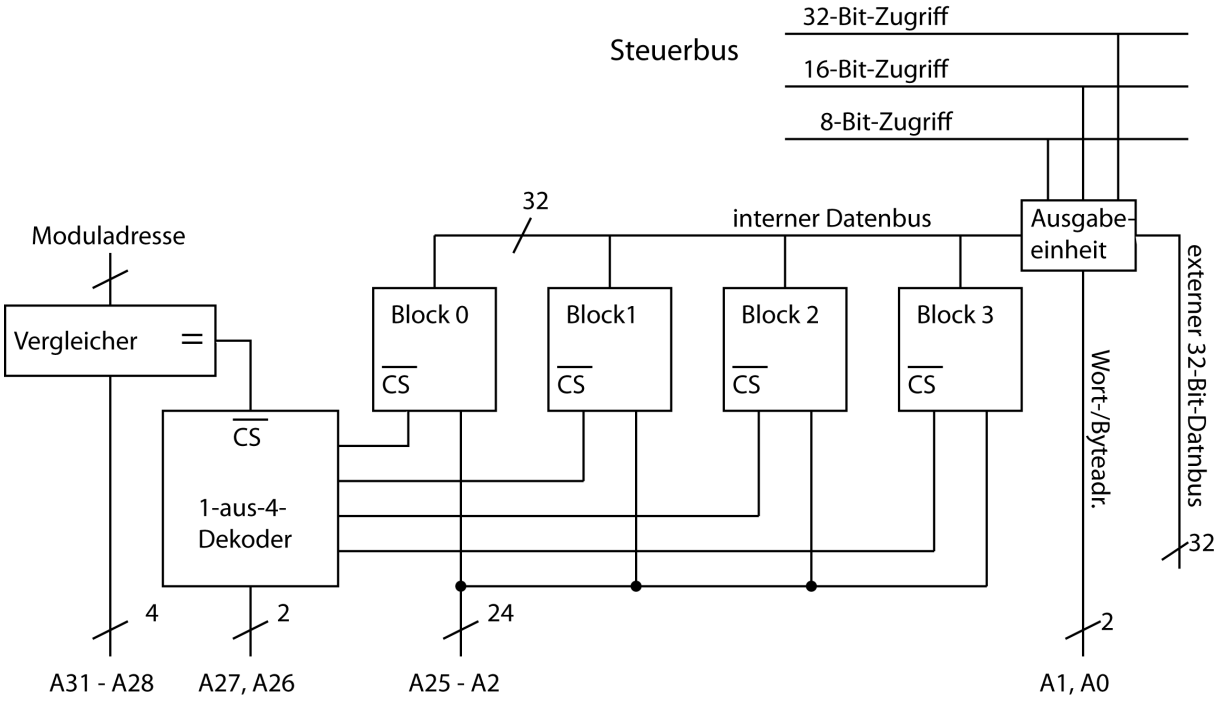


Figure 18.16: Aus mehreren Blöcken (Speicherbausteinen) kann ein Speichermodul von 256 MByte am 32-Bit-Datenbus aufgebaut werden.

Kenndaten:

- Jeder Block ist ein 16Mx32-Bit-Speicher. (Speicherchip oder Modul aus mehreren Chips)
- Das gesamte Modul hat also eine Kapazität von 64Mx32 Bit d.h. 256 MByte.
- Die Leitungen A_{31} – A_{28} werden für die Erkennung der Moduladresse verwendet.
- Es können somit maximal $2^4 = 16$ Module angesteuert werden.
- Die Leitungen A_{27}, A_{26} sind auf einen 1-aus-4-Dekoder geführt, der einen der vier Speicherblöcke freischaltet.
- Die Leitungen A_{25} – A_2 wählen innerhalb des Blocks eine von 16 M Adressen aus. Von dort werden in jedem Fall 32-Bit geladen.
- Die Leitungen A_1 und A_0 gehen an eine Ausgabeeinheit, die innerhalb des

32-Bit-Wortes ein, zwei oder vier Byte auswählt.

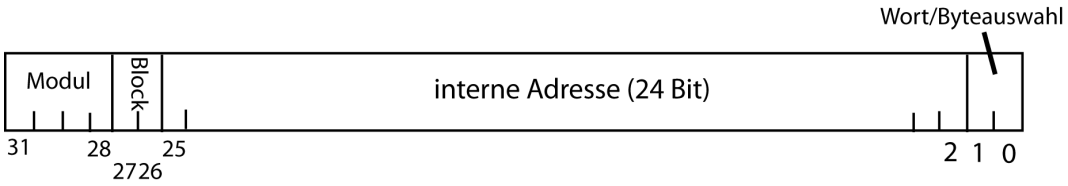


Figure 18.17: Das Adressaufteilungswort zu Abb.18.16.

Wir benennen die vier Byte einer 32-Bit-Einheit folgendermaßen:

Byte 3	Byte 2	Byte 1	Byte 0

Für 8- und 16-Bit Zugriffe sind die Adressleitungen A_1 und A_0 an einen Auswahlbaustein geführt. Dieser entnimmt aus dem 32-Bit-Datenpaket ein oder zwei Byte und übergibt sie auf den Datenbus. Die folgende Tabelle gibt einen Überblick (x=ignoriert):

Zugriffsart	A_1	A_0	Zugriff auf
32 Bit	x	x	Byte3 – Byte 0
16 Bit	0	x	Byte1 – Byte 0
16 Bit	1	x	Byte3 – Byte 2
8 Bit	0	0	Byte 0
8 Bit	0	1	Byte 1
8 Bit	1	0	Byte 2
8 Bit	1	1	Byte 3

Beispiel Es wird ein Byte an der Adresse 1A000027h angefordert. Aus der Aufteilung der binären Adresse 0001 1010 0000 0000 0000 0000 0010 0111b ergibt sich Modul=1, Block=2, Bitmuster an den Adresseingängen von Block2=800008h, laden von Systemadresse 2000024h. Von dort werden vier Byte geladen, also der Inhalt der Speicherzellen 1A000024h – 1A000027h. Die Bits 0 und 1 werden nun benutzt um aus diesem 32-Bit-Wort das richtige Byte auszuwählen, hier Byte 3.

Ausrichtung Die Leitungen A_1, A_0 sind nicht an den Speicherblock geführt. Das bedeutet, 32-Bit-Dateneinheiten werden automatisch an einer durch 4 teilbaren Adresse abgelegt und auch gelesen, die beiden letzten Adressbit werden ignoriert. Eine 32-Bit-Dateneinheit kann also mit *einem* Zugriff gelesen werden, aber nur, wenn sie an einer durch vier teilbaren Adresse beginnt. Das nennt man *Ausrichtung* (alignment). Eine fehlende Ausrichtung führt zumindest zu einem Zeitverlust, weil ein zweiter Zugriff nötig ist. (Bild 18.18)

Viele Systeme lassen nur ausgerichtete Daten zu und lösen bei fehlender Ausrichtung eine Ausnahme aus. Die Ausrichtung muss schon durch den Compiler vorgenommen werden, der über einen entsprechende Compileroption verfügt.

- 1-Byte-Dateneinheiten können überall liegen
- 2-Byte-Dateneinheiten beginnen immer an geraden Adressen
- 4-Byte-Dateneinheiten beginnen immer an durch 4 teilbaren Adresse.
- 8-Byte-Dateneinheiten beginnen immer an durch 8 teilbaren Adresse.

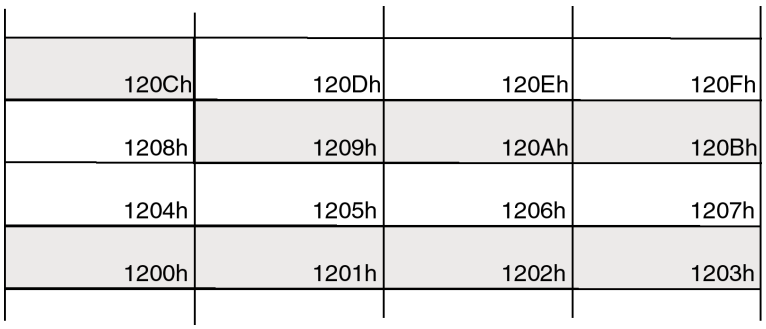


Figure 18.18: Die bei Adresse 1200h beginnende Dateneinheit ist 32-Bit-ausgerichtet. Die bei Adresse 1209h beginnende Dateneinheit ist nicht ausgerichtet, das System braucht zwei Zugriffe um dieses Datum zu laden oder bricht ab.

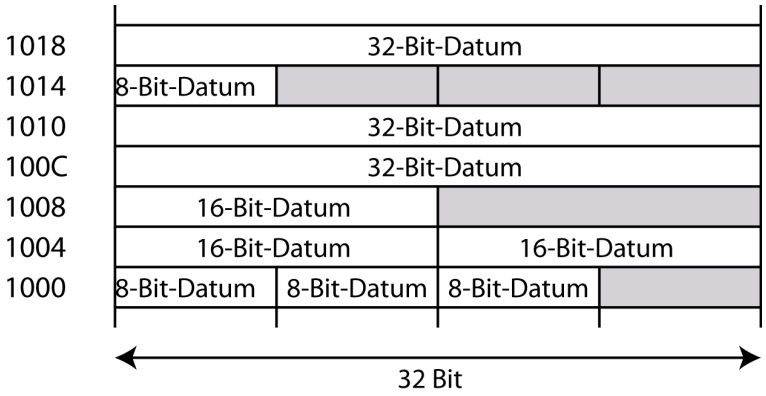


Figure 18.19: Variablen verschiedener Größe ausgerichtete im Speicher.

Übung: 18.3 Speicheraufbau

In einem Mikroprozessorsystem wird unter jeder Adresse ein Byte gespeichert; der Speicheraufbau ist durch folgendes Adresswort gekennzeichnet:

Bank-Nr.	Chip–Nr.	Interne Adresse auf Chip
29 28	27 25	24 0

A) Wie groß ist die Speicherkapazität jedes Chips?
B) Wieviele Speicherchips werden pro Bank betrieben?
C) Wie viele Speicherbänke können betrieben werden?
D) Wie groß ist die maximale gesamte Speicherkapazität?
E) Welchen Adressbereich belegt die Speicherbank Nr. 3?

Übung: 18.4 Systemadressen eines Speicherbausteins

Der Adressbereich eines Speicherbausteins beginnt bei Adresse 18000h (niedrigste Systemadresse), der Speicherbaustein hat 16 Adresseingänge. Wie groß ist die Speicherkapazität des Bausteines und welches ist die letzte Systemadresse (höchste Systemadresse), die auf diesem Baustein liegt?

Freie Adressleitungen

Der Systemadressraum bietet insgesamt Platz für 2^n Adressen. In den meisten Fällen ist der Speicher nicht voll ausgebaut, d.h. nicht der ganze Systemadressraum ist mit Bausteinen bestückt. Am höchstwertigen Ende des Adressbusses können auch Leitungen unbenutzt bleiben, dann gilt:

$$k + l < n$$

Freie Adressleitungen führen immer zu Mehrdeutigkeiten, da Unterschiede auf diesen freien Leitungen nicht ausgewertet werden. Es ergibt sich also, dass der gleiche Baustein auf mehreren Adressen angesprochen werden kann, die sich gerade in den nicht ausgewerteten Bitstellen unterscheiden. (Spiegeladressen).

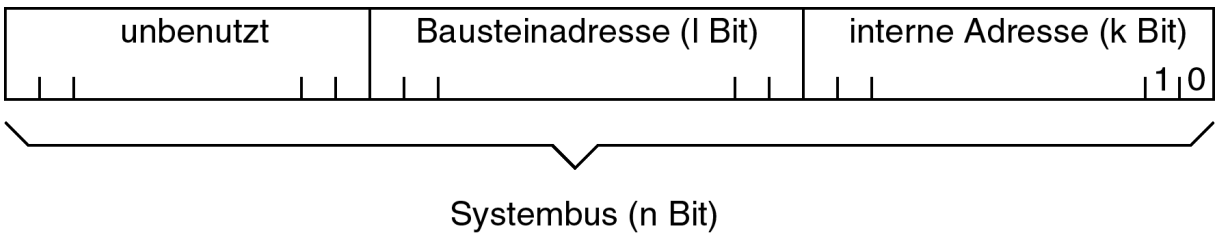
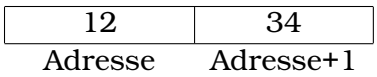


Figure 18.20: Das Adresswort beim Busanschluss von Bausteinen, wenn Adressleitungen frei bleiben.

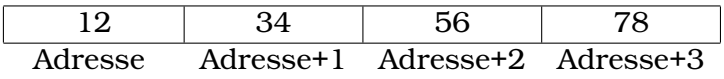
Big-Endian- und Little-Endian-Byteordnung

Es verbleibt noch die Frage, in welcher Reihenfolge sollen die Bytes größerer Datenstrukturen im Speicher abgelegt werden? Diese Frage klingt banal, ist aber von großer Bedeutung! Die Zugspitze hat eine Höhe von 2964 m. Speichert man dies als 16-Bit-Zahl ab und setzt die beiden Bytes später in umgekehrter Reihenfolge wieder zusammen, so ist die Zugspitze plötzlich 37899 m hoch!

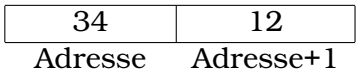
Die erste Möglichkeit ist, die Abspeicherung mit dem höchstwertigen Byte zu beginnen. Diese Byteordnung heißt *big-endian*. Mit big-endian-Byteordnung wird z.B. die Zahl 4660d = 1234h im 16-Bit-Format im Speicher folgendermaßen abgelegt:



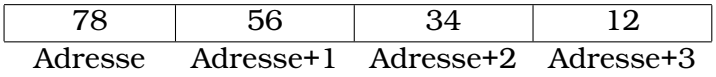
Die Zahl 12345678h im 32-Bit-Format wird mit big-endian-Byteordnung so im Speicher abgelegt:



Die Alternative ist die *little-endian*-Byteordnung, dabei wird mit dem niedrigstwertigen Byte begonnen. Die gleichen Zahlen liegen nun ganz anders im Speicher. Die Zahl 4660d = 1234h im 16-Bit-Format in little-endian-Byteordnung ist so abgelegt:



Die Zahl 12345678h im 32-Bit-Format und little-endian-Byteordnung wird so abgelegt:



Verwendung:

- SPARC-Rechner und IBM-Großrechner: big-endian
- PCs: little-endian

Von Bedeutung in Netzwerken, die Rechnerwelten verbinden

Übung: 18.5 Endianess

Die Zugspitze hat eine Höhe von 2964 m. Welche Höhe ergibt sich, wenn man dies als 16-Bit-Zahl abspeichert und die beiden Bytes später in der falschen Reihenfolge wieder zusammensetzt?

Speicherbezogene und isolierte E/A-Adressierung

Speicherbezogene E/A-Adressierung (memory mapped I/O-Adressing) Es gibt nur einen Systemadressraum, den sich Speicherbausteine und E/A-Bausteine teilen. Ein Teil der Speicheradressen ist auf I/O-Bausteine geleitet, so dass diese im Adressraum des Speichers erscheinen (Abb. 18.21). (z.B. Motorola)

Isolierte E/A-Adressierung (isolated I/O-Adressing) , es wird über ein zusätzliches Steuersignal, das z.B. $MEM/\overline{IO}$ heißen kann, von der CPU mitgeteilt, ob ein Speicher oder E/A-Baustein angesprochen werden soll (Abb. 18.22). Isolierte Adressierung wird z.B. von Intels 80x86-Prozessoren verwendet, die Assembler-Befehle für die Ein- und Ausgabe lauten `IN` und `OUT`.

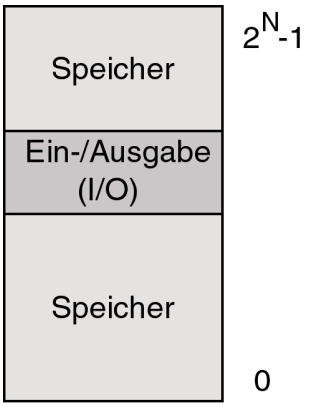


Figure 18.21: Der Adressraum bei speicherbezogener Ein-/Ausgabeadressierung.

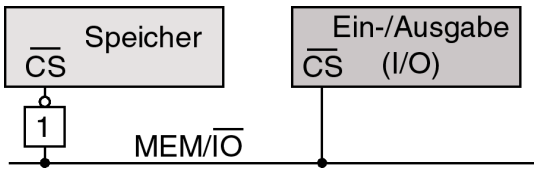


Figure 18.22: Für die isolierte Adressierung wird eine zusätzliche Busleitung gebraucht.

Synchrone und asynchrone Busse

Der Zeitablauf für die vielen Vorgänge auf dem Bus, muss absolut zuverlässig geregelt sein. Es darf z.B. nie vorkommen, dass an den Prozessor falsche Daten übermittelt werden, weil ein Speicher- oder E/A-Baustein seine Daten zu spät auf den Bus gelegt hat. Für dieses Problem gibt es zwei grundsätzliche Ansätze: Den synchronen und den asynchronen Bus.

Beim *synchronen Bus* gibt es auf einer separaten Taktleitung ein Taktsignal. Dieses Taktsignal ist durch einen Schwingquarz stabilisiert und bildet das Zeitraster für alle Vorgänge auf dem Bus. Das Taktsignal ist normalerweise ein symmetrisches Rechtecksignal mit der Taktfrequenz f (Abb. 18.24). Das sich wiederholende Sig-

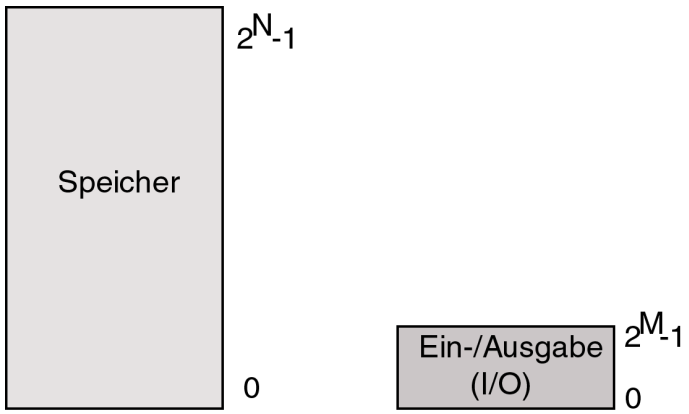


Figure 18.23: Der Adressraum bei isolierter Adressierung.

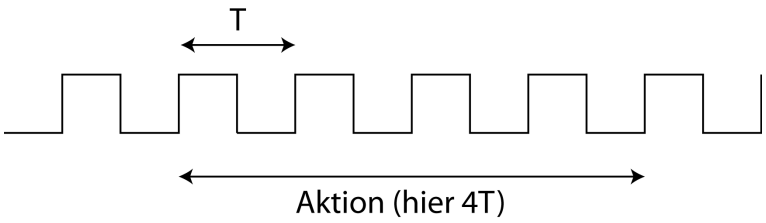


Figure 18.24: Ein synchroner Bus wird durch ein Taktsignal synchronisiert.

nalstück des Taktsignals – LOW-Phase plus HIGH-Phase – nennt man einen *Taktzyklus* oder einfach einen *Takt*. Die Dauer eines Taktes ist die Taktzykluszeit $T = 1/f$. Heutige Computerbusse laufen mit Taktfrequenzen von mehr als 100 MHz. Bei 100 MHz dauert ein Taktzyklus $T = 1/10^8\ s^{-1} = 10\ ns$.

Am synchronen Bus ist die Dauer der Aktionen ein ganzzahliges Vielfaches der Taktzeit. Es wird also immer aufgerundet, was zu einer gewissen Zeitverschwendung führt. Der asynchrone Bus vermeidet die Nachteile des synchronen, er kennt kein festes Zeitraster, sondern arbeitet mit Quittungssignalen und setzt fort, sobald ein Datentransfer quittiert ist.

Kurz:

Synchroner Bus Gemeinsames Taktsignal, alle Bausteine sind darauf synchronisiert, alle Zeiten werden aufgerundet auf ganzzahlige Vielfache des Taktzyklus, Wartezeiten, Waitstates, verschiedene Taktfrequenzen für verschiedene Busse.

Asynchroner Bus Kein Taktsignal, Bausteine quittieren alle Datentransfers mit einem eigenen Quittungssignal, keine Zeit wird verschenkt, zusätzliche Leitungen.

Busdesign

Die wichtigsten Design-Parameter für ein Bussystem sind

- Taktfrequenz des Busses
- Breite des Adressbusses, daraus folgt der Adressraum

- Breite des Datenbusses, mit dem Takt folgt daraus der maximale Datendurchsatz
- Multiplexing Ja/Nein

19 Rechnerarchitekturen

19.1 Interner Aufbau eines Mikroprozessors

Alle Mikroprozessoren bestehen in ihrem Inneren aus mehreren Baugruppen, die für verschiedene Aufgaben zuständig sind (siehe Abb. 19.1).

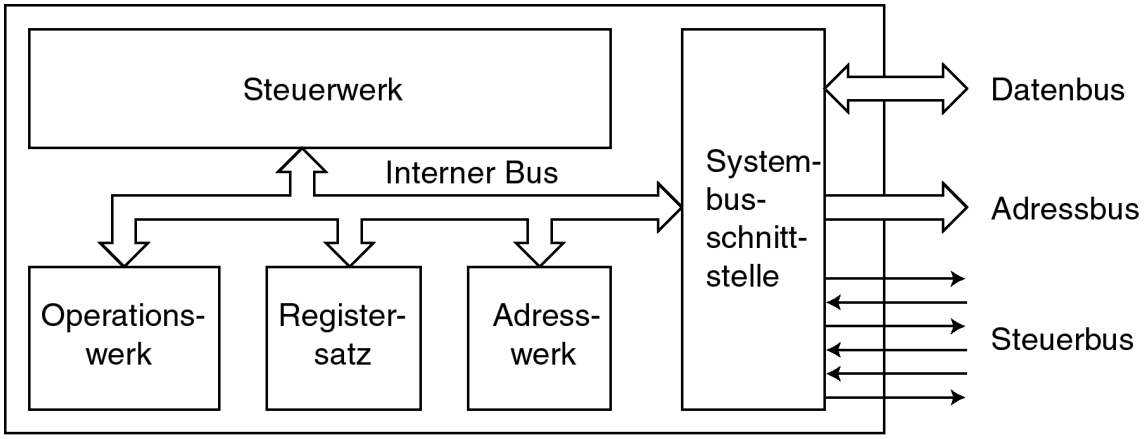


Figure 19.1: Interner Aufbau eines Mikroprozessors (stark vereinfacht)

Der Registersatz enthält einen Satz von Registern, mit dem Daten innerhalb des Prozessors gespeichert werden können. Ein Register ist eine Gruppe von Flipflops mit gemeinsamer Steuerung.

Das Operationswerk führt die eigentliche Verarbeitung, d.h. die logischen und arithmetischen Operationen, an den übergebenen Daten aus.

Das Steuerwerk ist verantwortlich für die Ablaufsteuerung sowohl im Inneren des Prozessors als auch im restlichen System.

Das Adresswerk erzeugt die erforderlichen Adressen, um auf Daten und Code im Hauptspeicher zugreifen zu können.

Die Systembus-Schnittstelle enthält Puffer- und Treiberschaltungen, um den Datenverkehr über den Systembus abzuwickeln.

Registersatz

Ein Register ist eine Gruppe von Flipflops mit gemeinsamer Steuerung, jedes Flipflop speichert 1 Bit. Die Register stellen prozessorinterne Speicherplätze dar und sind am internen Datenbus des Prozessors angeschlossen. Die Steuerung kann nun dafür sorgen, dass Daten vom internen Datenbus in ein Register eingeschrieben

werden oder vom Register auf den internen Datenbus ausgegeben werden (Abb. 19.2). Alle Mikroprozessoren enthalten mehrere Register, die Breite ist meist 8, 16,

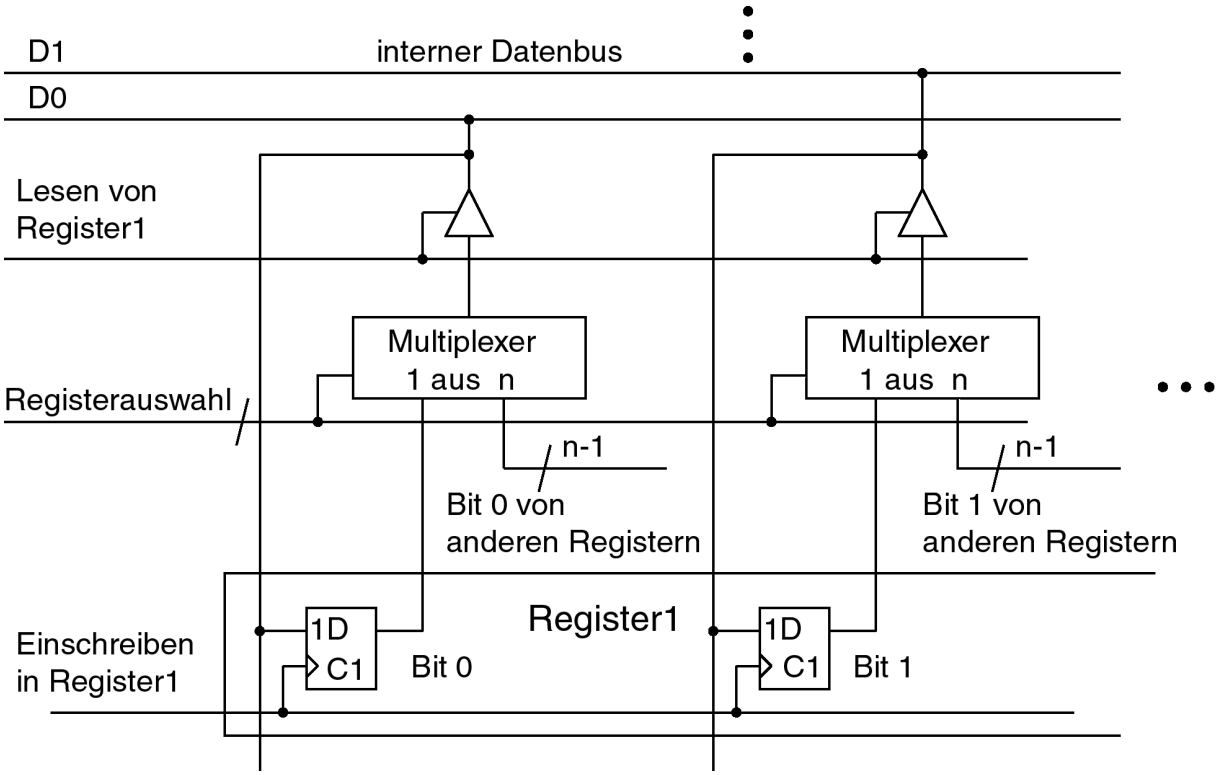


Figure 19.2: Prinzipieller Aufbau eines Satzes von n Registern in einem Prozessor. Bei den Flipflops ist 1D der Dateneingang und C1 der Clockeingang, der die Datenübernahme von 1D triggert. Es sind nur Bit 0 und Bit 1 gezeichnet, alle weiteren sind entsprechend beschaltet.

32, 64 oder 128 Bit, d.h. Flipflops. Es gibt Register, die vom Programm in Maschinenbefehlen direkt angesprochen werden können. (*Registersatz*). Manche Register werden vom Mikroprozessor intern benutzt.

Verwendung der Register:

Universalregister werden zur kurzzeitigen Zwischenspeicherung von Daten benutzt; Zugriff ist schneller als auf Speicher.

Spezialregister sind auf Grund der internen Verschaltung nur für ganz bestimmte Zwecke vorgesehen. Beispiele: Programmzähler und Stackpointer, Speicherverwaltungsregister

Maschinenstatusregister und das Maschinensteuerregister Jedes Flipflop hat hier eine ganz eigene Bedeutung und auch eine separate Steuerung, so genannte *Flags*.

Register von Peripheriebaugruppen Sind meist in Mikrocontrollern enthalten, für Steuerung, Daten und Status von Peripheriegruppen, wie z.B. Schnittstellen.

Steuerwerk

Aufgaben des Steuerwerks (*Control Unit*):

- Dekodierung der Opcodes
- Erzeugung der internen Signale für die Ausführung des Befehles
- Ansteurung der Busschnittstelle zur Erzeugung der externen Signale für die Ausführung des Befehles
- Auswertung von internen Statussignalen (z.B. Carry-Flag)
- Auswertung von externen Statussignalen (z.B. Interrupt-Request)

Beispiel Der Programmzähler zeigt auf einen Speicherplatz, an dem der Opcode des Befehles *Kopiere Register 1 in Register 2* liegt. Die erzeugten Steuersignale können wie folgt beschrieben werden:

1. Programmzähler auf den Adressbus legen.
2. Aktivierung der externen Steuerleitungen für Lesezugriff im Speicher.
3. Einspeicherimpuls für Befehlsregister erzeugen, Opcode von Datenbus entnehmen und im Befehlsregister einspeichern.
4. Dekodierung des Opcodes.
5. Register 1 auf Senden einstellen und auf internen Datenbus aufschalten.
6. Register 2 auf internen Datenbus aufschalten, nicht auf Senden einstellen.
7. Einspeicherimpuls an Register 2 geben.
8. Programmzähler inkrementieren.

Kommentare: 1. Die ersten drei Schritte bilden den Befehlslesezyklus. (ohne evtl. Wartezeiten)
2. In diesem Beispiel Ausführung des Befehles innerhalb des Mikroprozessors ohne Buszugriff

Das Steuerwerk hat ziemlich komplizierte Sequenzen von Signalen zu erzeugen, und viele Signale zu berücksichtigen. Wie kann das realisiert werden?

1. Abspeicherung der auszugebenden Bitmuster in einem ROM, Abruf von dort über Sequenzen und Verzweigungen, die die richtigen Adressen ansprechen; aus dieser Grundidee entstammen die CISC-Prozessoren.
2. Reine einschrittige, schnelle Digitallogik; diesem Lösungsansatz entstammen die RISC-Prozessoren.

Erzeugung einer Signalsequenz für einen Maschinenbefehl mit dieser Schaltung:

- Ein Teil der Steuersignale ist an die Adresseingänge des ROMs zurückgeführt.
- Ein Taktsignal (Prozessortakt) steuert, dass das ROM zyklisch immer wieder ausgelesen wird.
- Durch jedes aus einer Speicherzelle ausgelesene Bitmuster wird die Adresse für den nächsten Lesevorgang mit bestimmt.
- Von der neuen Adresse wird ein anderes Bitmuster ausgelesen, es ergibt sich wieder eine neue Adresse
- Es können beliebig lange Sequenzen an Ausgangssignalen erzeugt werden.

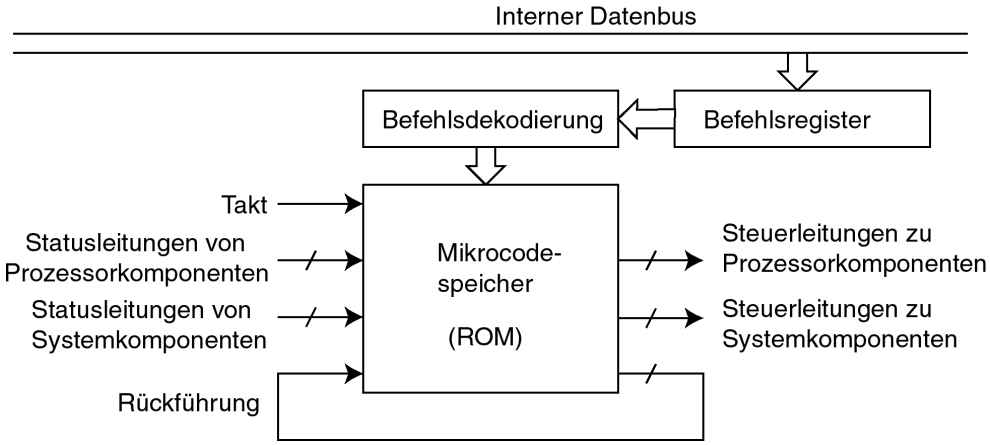


Figure 19.3: Steuerwerk eines einfachen Mikroprozessors. Anmerkung: RISC-Prozessoren verwenden statt des Mikrocode-Speichers eine schnelle digitale Logikschaltung.

- Wiederholungen und Verzweigungen sind möglich
- Alle Elemente einer Programmierung sind möglich
- Diese Art der Steuerung heißt *Mikroprogrammierung*
- Die Bitmuster im ROM heißen *Mikrocode*, das ROM heißt *Mikrocode-ROM*

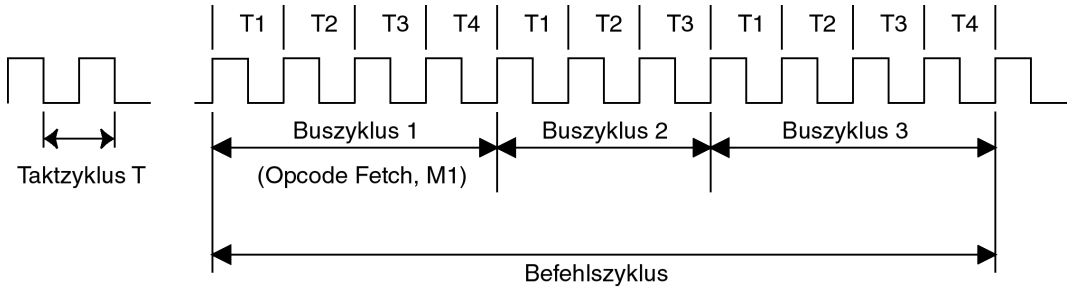


Figure 19.4: Ein Befehlszyklus mit drei Maschinenzyklen, die jeweils mehrere Taktzyklen umfassen.

Das Operationswerk (Rechenwerk)

Der zentrale Teil des Operationswerkes ist die *arithmetisch/logische Einheit*, ALU (arithmetic and logical unit).

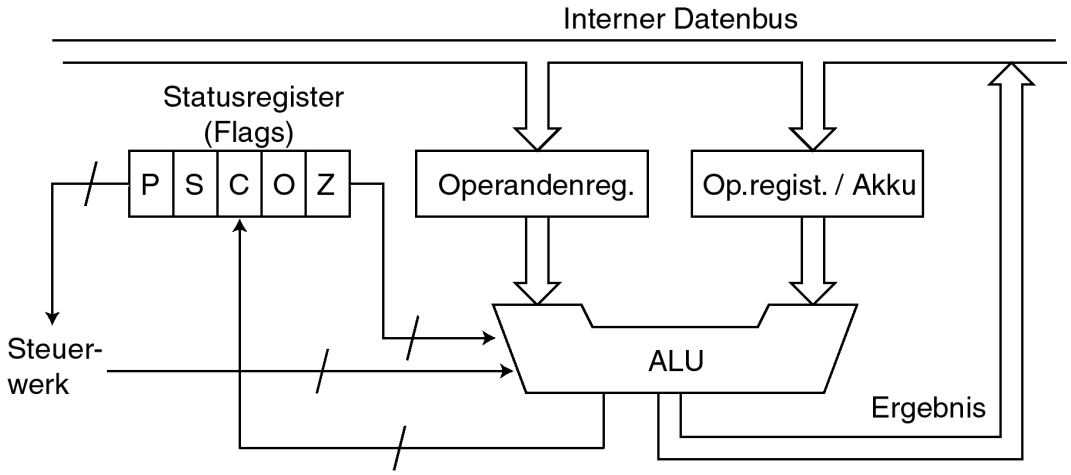


Figure 19.5: Zentraler Teil des Operationswerkes.

- Die ALU kann über Steuereingänge auf eine Vielzahl von arithmetischen und logischen Operationen ("'Modes"')eingestellt werden
- Diese Einstellung nimmt das Steuerwerk nach der Dekodierung vor.
- Die ALU selbst hat keine Speichereigenschaften, deshalb sind vor die Eingänge Operandenregister geschaltet.
- Das Ergebnis kann vom Ausgang über entsprechende Busschaltung sofort wieder in ein Operandenregister geladen werden
- Das Steuerwerk hat so die Möglichkeit, komplexere Operationen algorithmisch aus mehreren ALU-Operationen aufzubauen;

Die letzte Möglichkeit ist typisch für CISC-Prozessoren:

- CISC-Prozessoren haben meistens einen Multiplikationsbefehl, obwohl die ALU keine Multiplikation ausführen kann.
- Der Multiplikationsbefehl algorithmisch realisiert, indem er aus mehreren ALU-Operationen – z.B. Additions- und Schiebebefehlen – zusammmengesetzt wird
- Solche Befehle brauchen viele Takte, sind also langsam (Bsp Intel-80386, Addition zweier Register 2 Takte, eine Multiplikation bis zu 38 Takte.)

Hinweis: bei RISC-Prozessoren ist das anders!

Die gewünschte Bitbreite erreicht man durch Parallelschaltung mehrerer 74181. Damit dann ein Übertrag weitergereicht werden kann, gibt es einen Ausgang für einen Übertrag vom höchstwertigen Bit an die nächste ALU (Carry-Out) und einen Eingang für die Aufschaltung eines Übertrages von der vorigen ALU auf das niedrigstwertige Bit (Carry-In).

Table 19.1: Funktionstabelle der arithmetisch/logischen Einheit 74181 für positive Logik. Die Daten-Eingangssignale sind A und B mit je vier Bit sowie der Carry-Eingang $\overline{C_n}$ mit 1 Bit. Die Ausgangssignale sind die vier Ergebnis-Leitungen sowie der Carry-Ausgang. Mit dem Mode-Eingang M wird die Betriebsart gewählt: M=H logische Funktionen, M=L arithmetische Funktionen. Über die Steuereingänge S_0, S_1, S_3, S_4 wird die gewünschte Operation ausgewählt. Verknüpfungssymbole: $\vee$ bitweise logisches ODER, $\wedge$ bitweise logisches UND, $\neq$ bitweise logische Antivalenz, shl verschieben nach links.

Steuersignale				**Ergebnis am Ausgang der ALU**		
				M=H logische	**M=L arithmetische Operationen**	
S_3	S_2	S_1	S_0	**Operationen**	$\overline{C_n} = H$ (ohne Carry)	$\overline{C_n} = L$ (mit Carry)
L	L	L	L	$\overline{A}$	A	$A+1$
L	L	L	H	$\overline{A \vee B}$	$A \vee B$	$(A \vee B)+1$
L	L	H	L	$\overline{A} \wedge B$	$A \vee \overline{B}$	$(A \vee \overline{B})+1$
L	L	H	H	0	-1	0
L	H	L	L	$\overline{A \wedge B}$	$A + A \wedge \overline{B}$	$A + A \wedge \overline{B} + 1$
L	H	L	H	$\overline{B}$	$(A \vee B) + A \wedge \overline{B}$	$(A \vee B) + A \wedge \overline{B} + 1$
L	H	H	L	$A \neq B$	$A - B - 1$	$A - B$
L	H	H	H	$A \wedge \overline{B}$	$A \wedge \overline{B} - 1$	$A \wedge \overline{B}$
H	L	L	L	$\overline{A} \vee B$	$A + A \wedge B$	$A + A \wedge B + 1$
H	L	L	H	$\overline{A \neq B}$	$A + B$	$A + B + 1$
H	L	H	L	B	$(A \vee \overline{B}) + A \wedge B$	$(A \vee \overline{B}) + A \wedge B + 1$
H	L	H	H	$A \wedge B$	$A \wedge B - 1$	$A \wedge B$
H	H	L	L	1	$A + (A\ shl\ 1)$	$A + A + 1$
H	H	L	H	$A \vee \overline{B}$	$(A \vee B) + A$	$(A \vee B) + A + 1$
H	H	H	L	$A \vee B$	$A \vee \overline{B} + A$	$A \vee \overline{B} + A + 1$
H	H	H	H	A	$A - 1$	A

Betrachten wir einige Beispiele für Maschinenbefehle an einem hypothetischen Mikroprozessor, der die dargestellte ALU 74181 enthält und entnehmen die notwendige Ansteuerung aus Tabelle 19.1.

Beispiel 1 Ein Register soll inkrementiert werden. Das Steuerwerk bringt den Registerinhalt an den A-Eingang und die ALU erhält die Steuerungssignale M=L, $\overline{C_n}$=L, S_3=L, S_2=L, S_1=L, S_0=L, das Ergebnis wird über den internen Bus wieder in das Ursprungsregister geladen.

Beispiel 2 Ein Register soll mit einem zweiten durch ein bitweise logisches UND verknüpft werden, das Ergebnis soll im ersten der beiden Register abgelegt werden. Das Steuerwerk bringt den Inhalt der beiden Register an den A- und B-Eingang, die ALU erhält die Steuerungssignale M=H, S_3=H, S_2=L, S_1=H, S_0=H, das Ergebnis wird über den internen Bus wieder in das erste der beiden Register geladen.

Statusregister und Flags

- Das Statusregister (auch Zustandsregister, condition code register) besteht aus Einzel-Flipflops

- Diese werden Flags genannt
- Jedes Flag hat eine ganz bestimmte Bedeutung
- Die Flags werden bei Operationen der ALU gesetzt und speichern Informationen über den Verlauf der letzten ALU-Operation
- Die Flags steuern manche Verzweigungen bei der Ausführung eines Befehls; Beispiel: Ein bedingter Sprungbefehl "'Jump if Zero"' wird z.B. nur dann ausgeführt, wenn das Ergebnis der letzten Operation Null war, erkennbar am gesetzten Zero-Flag.
- Manche Flags werden intern benutzt, z.B. Carry-Flag

Achtung: Bei Flags hat sich eine spezielle Namensregelung eingebürgert:

- Flag löschen (to clear) heißt eine '0' eintragen
- Flag setzen (to set) heißt eine '1' eintragen
- Ein gelöschtes Flag enthält eine '0'
- Ein gesetztes Flag enthält eine '1'

Das **Zero Flag** (Null-Flag, Nullbit) wird vom Operationswerk gesetzt, wenn das Ergebnis der letzten Operation gleich Null war; wenn nicht, wird das Zero Flag gelöscht. Damit kann man bequem Schleifen programmieren: Man dekrementiert einen Schleifenzähler und verlässt die Schleife mit einem bedingten Sprungbefehl, wenn er Null ist.

Das **Carry Flag** (Übertragsbit) zeigt bei der Addition einen Übertrag aus dem MSB heraus auf das (nicht mehr vorhandene) nächst höherwertige Bit an. Bei Subtraktion zeigt es ein Borgen von dem nächst höherwertigen Bit auf das MSB an. Dies wird beim Rechnen mit vorzeichenlosen Zahlen ausgenutzt.

Das **Overflow Flag** (Überlaufbit) zeigt einen bei Addition oder Subtraktion entstehenden Übertrag auf das MSB an. Da das MSB bei Zweierkomplement-Zahlen das Vorzeichen enthält, wird das Überlaufbit beim Rechnen mit Vorzeichen gebraucht (Zweierkomplementzahlen).

Das **Sign Flag** (Vorzeichenbit) gibt das entstandene MSB wieder. Bei Zweierkomplement-Zahlen entspricht dieses genau dem Vorzeichen, ist also gesetzt, wenn das Ergebnis negativ ist.

Das **Parity Flag** (Paritätsbit) zeigt an, ob die Anzahl der '1'-Bits im Ergebnis gerade oder ungerade ist.

Adresswerk und Adressierungsarten

Viele Maschinenbefehle müssen mit Daten umgehen, die an verschiedenen Orten zu finden sind: Im Hauptspeicher, in Registern oder im Maschinencode. Oft muss zur Laufzeit berechnet werden, auf welchen Speicherplatz zugegriffen wird, z. B.

bei Arrays. Die Maschinenbefehle bieten daher verschiedene Möglichkeiten, diese Daten aufzufinden, die *Adressierungsarten*.

Unmittelbare Adressierung Das Datum steht als Direktoperand im Maschinencode hinter dem Opcode.

Registeradressierung Das Datum steht in einem Register

Direkte Speicher-Adressierung Das Datum steht im Speicher, die Adresse ist konstant

Register-indirekte Speicher-Adressierung Das Datum steht im Speicher, die Adresse steht in einem Register

Speicher-indirekte Speicher-Adressierung Das Datum steht im Speicher, die Adresse steht im Speicher (selten)

Bei der Speicheradressierung berechnet das Adresswerk die letztlich über den Adressbus ausgegebene Adresse (effektive Adresse). Da die Register-indirekte Adressierung wichtig und häufig ist, werden für diese Berechnung meistens mehrere Varianten angeboten:

- Zum Inhalt eines Registers kann eine Konstante (Displacement) addiert werden. Anwendung: Bei der Adressierung von Arrays hält die Konstante die Adresse des ersten Elementes.
- Zum Inhalt eines Registers kann der Inhalt eines zweiten Registers addiert werden. Anwendung: Zweidimensionle Arrays
- Nach der Adressberechnung wird eines der beteiligten Register automatisch erhöht (Autoinkrement) oder erniedrigt (Autodekrement), Anwendung: Arrayzugriff in Schleifen.
- Der Inhalt eines Adressregisters wird vor der Adressberechnung mit 2, 4 oder 8 mal genommen (Skalierung); Anwendung: Adressierung von Feldern mit Elementen von 2, 4 oder 8 Byte.

Beispiel Das Basisregister enthält den Wert 0020h, ein Displacement von 300h wird addiert. Der Zugriff erfolgt auf Speicheradresse 0320h.

Beispiel Das Indexregister enthält nacheinander die Werte 0010h, 0011h, 0012h und 0013h. Ein Displacement von 300h wird addiert, der Skalierungsfaktor beträgt 4. Der Zugriff erfolgt auf die Speicheradressen 0340h, 344h, 348h und 34Ch.

Beispiel Das Basisregister enthält den Wert 1000h, das Indexregister enthält nacheinander die Werte 0020h, 21h, 22h. Ein Displacement von 300h wird addiert, der Skalierungsfaktor beträgt 2. Der Zugriff erfolgt auf die Speicheradressen 1340h, 1342h, 1344h.

Die Systembus-Schnittstelle

Die Systembus-Schnittstelle treibt die externen Busleitungen und stellt damit die Verbindung zur Außenwelt (Speicher- und E/A-Bausteine) dar. Die Bustreiber sind meist als Tristate-Ausgänge konstruiert (HIGH, LOW, floating) Die Busse werden wie folgt angesteuert:

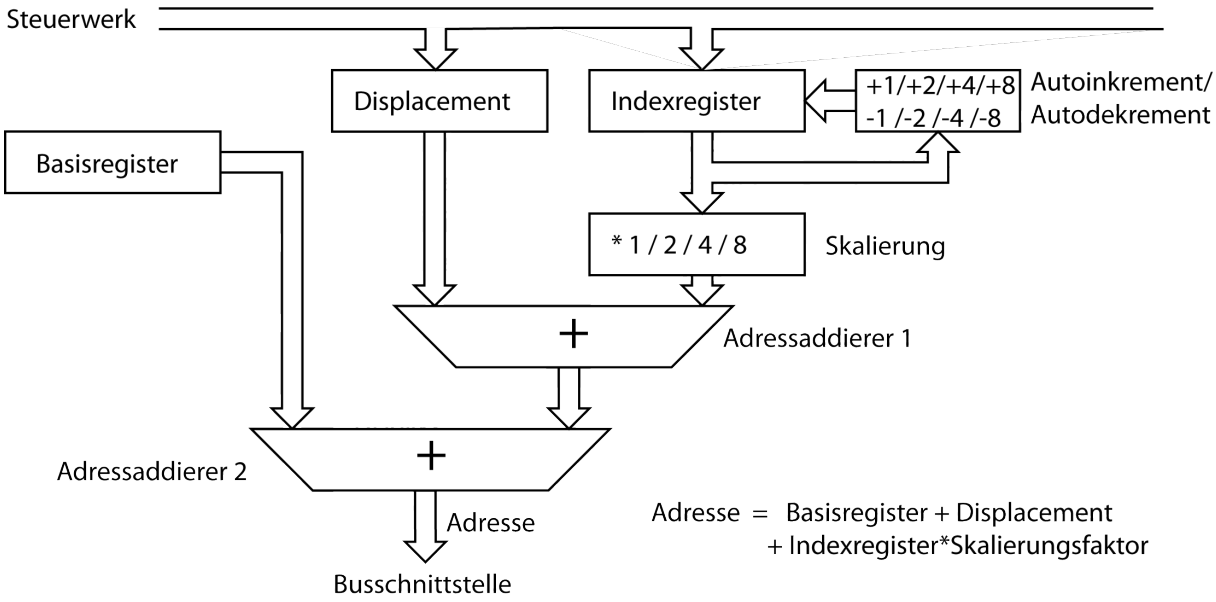


Figure 19.6: Ein Adressrechner mit Autoinkrement/Autodekrement sowie Skalierung für das Indexregister.

- Die Adressleitungen werden unidirektional betrieben (nur schreiben)
- Die Datenleitungen werden bidirektional betrieben (schreiben und lesen)
- Der Steuerbus ist inhomogen (nur lesen, nur schreiben oder schreiben und lesen)

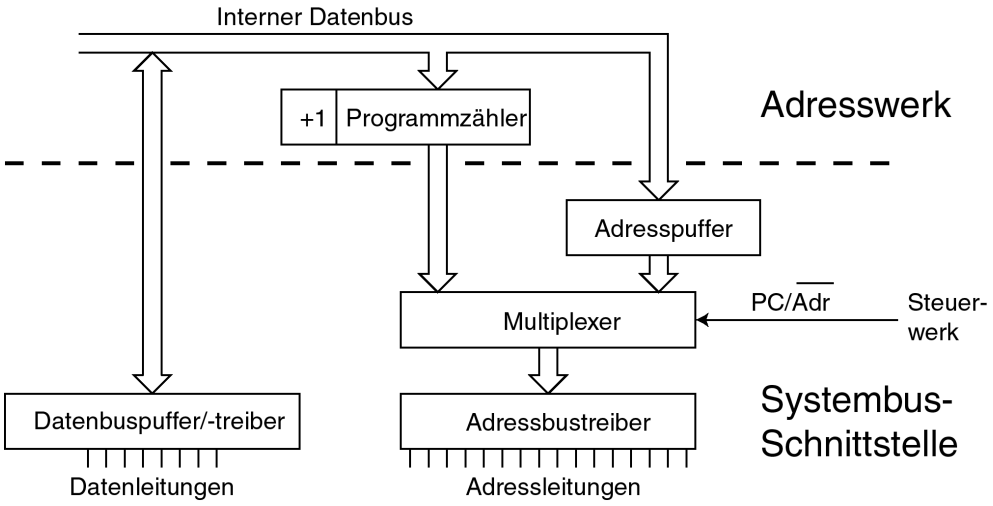


Figure 19.7: Die Systembus-Schnittstelle. Beispielhaft sind hier 8 Datenleitungen und 16 Adressleitungen gezeichnet.

Der *Datenbus-Puffer* ist ein bidirektional betriebenes Pufferregister, es speichert alle Daten, die vom Prozessor an den Bus ausgegeben oder vom Bus empfangen werden. Die Zwischenspeicherung in Pufferregistern ist dient auch der Synchroni-

Table 19.2: Adressierungsarten für Speicherzugriffe. Die Displacements werden als Operanden im Maschinencode mitgeführt.

Direkte Adressierung	
	Adresse wird gebildet aus
Direkte Adressierung	konstanter Ausdruck

Registerindirekte Adressierung	
Variante	*Adresse wird gebildet aus*
Basisadressierung	Basisregister Basisregister + Displacement
Indexadressierung	Indexregister Indexregister + Displacement
Basis-indizierte Adressierung	Basisregister + Indexregister Basisregister + Indexregister + Displacement

sation: Der Prozessor läuft mit höherem Takt als der Speicherbus.

19.2 CISC-Architektur und Mikroprogrammierung

Mikroprogrammierung bietet beim Entwurf von Prozessoren viele Vorteile:

Flexibilität Dem Prozessorbefehlssatz können auf Software-Ebene neue Befehle hinzugefügt werden. Das macht es leichter, den Prozessor weiter zu entwickeln und an die Bedürfnisse des Marktes anzupassen.

Fehlerbeseitigung Design-Fehler können durch Einspielen eines neuen Mikrocodes sogar noch beim Kunden behoben werden.

Kompatibilität und Emulation Bei neuen Prozessorkonzepten kann auf Software-Ebene der Befehlssatz von Vorgängern nachgebildet und dadurch Kompatibilität hergestellt werden. Sogar die Emulation anderer Prozessoren ist möglich.

Varianten Es können leicht Varianten von Prozessoren mit anderen Befehlssätzen – z.B. Mikrocontroller – hergestellt werden. Sogar Änderungen am Befehlssatz nach Kundenwunsch sind möglich.

Die Folgen waren:

- Der Befehlssatz der mikroprogrammierten Prozessoren wurde immer größer und umfasste in den achtziger Jahren oft mehrere hundert Befehle
- Da die einfachen Befehle zuerst entworfen wurden, kamen immer kompliziertere Befehle dazu.
- Prozessoren mit Mikroprogrammierung und komplexem Befehlssatz heißen *Complex Instruction Set Computer*, kurz *CISC*
- Lösung bis in die 70-er Jahre

Krise der CISC-Prozessoren – die Nachteile wurden allmählich spürbar:

- Dekodierung der vielen komplexen Befehle wurde immer aufwändiger
- Die Dekodierungseinheit brauchte zunehmend Zeit und auch Platz auf dem Chip.
- Wahrscheinlichkeit von Entwurfsfehlern im Steuerwerk stieg an
- Entwicklung von Hochsprachen-Übersetzern wird immer komplizierter

19.3 RISC-Architektur

Bei der RISC-Architektur versucht man die Nachteile der CISC-Prozessoren zu vermeiden und performante, schnelle Prozessoren zu bauen. Das sollte vor allem durch einen kleinen Satz einfacher Befehle gelingen. RISC=*Reduced Instruction Set Computer*. Man hat folgende Prinzipien formuliert:

Skalarität Es soll möglichst mit jedem Takt ein Befehl bearbeitet werden. Dieses Ziel ist sehr weitreichend und erfordert aufwändige konstruktive Maßnahmen nach sich. Prozessoren, die mehr als einen Befehl pro Takt bearbeiten, heißen *superskalar*. Einfache RISC-Prozessoren erreichen nicht unbedingt Skalarität.

Verzicht auf Mikroprogrammierung Alle Befehle sind einer Hardwareeinheit zugeordnet ("'fest verdrahtet'"). Dadurch sind nur einfache Befehle möglich, die schnell ausgeführt und dekodiert werden können. Einen Multiplikationsbefehl wird es nur geben, wenn auch ein Hardware-Multiplizierer da ist.

Load/Store-Architektur Die Kommunikation mit dem Hauptspeicher wird nur über die Befehle Laden und Speichern (LOAD und STORE) abgewickelt. Dadurch wird der zeitkritische Transport zwischen Prozessor und Speicher auf ein Minimum beschränkt. ALU-Operationen können dementsprechend nur auf Register angewendet werden. Es gibt keine Befehle, die einen Speicheroperanden laden, bearbeiten und wieder speichern (Read-Modify-Write-Befehle), dies sind typische mikroprogrammierte Befehle.

Großer Registersatz Viele Register ermöglichen es, viele Variablen in Registern zu halten und damit zeitraubende Hauptspeicherzugriffe einzusparen. Üblich sind mindestens 16 Allzweck-Register, meistens deutlich mehr.

Feste Befehlswortlänge Alle Maschinenbefehle haben einheitliche Länge, das vereinfacht das Laden und Dekodieren der Befehle. Die Verlängerung des Codes, die durchaus 50% betragen kann, nimmt man in Kauf.

Horizontales Befehlsformat In den Maschinenbefehlen haben Bits an fester Position eine feste und direkte (uncodierte) Bedeutung; auch dies beschleunigt die Dekodierung.

Orthogonaler Befehlssatz Jeder Befehl arbeitet auch mit jedem Register zusammen.

Der Registersatz von RISC-Prozessoren

Ein RISC-Prozessor besitzt mindestens 32 gleichwertige, universelle Register, deren Organisation unterschiedlich sein kann. Das Ziel ist immer, Variablen möglichst in Registern zu behalten und Hauptspeicher- bzw. Cachezugriffe zu vermeiden. Im einfachsten Fall sind die Register als homogener Block organisiert (Abb. 19.8 links).

Ein schönes Beispiel für flexibel überlappende Registerfenster ist Infineons C167.

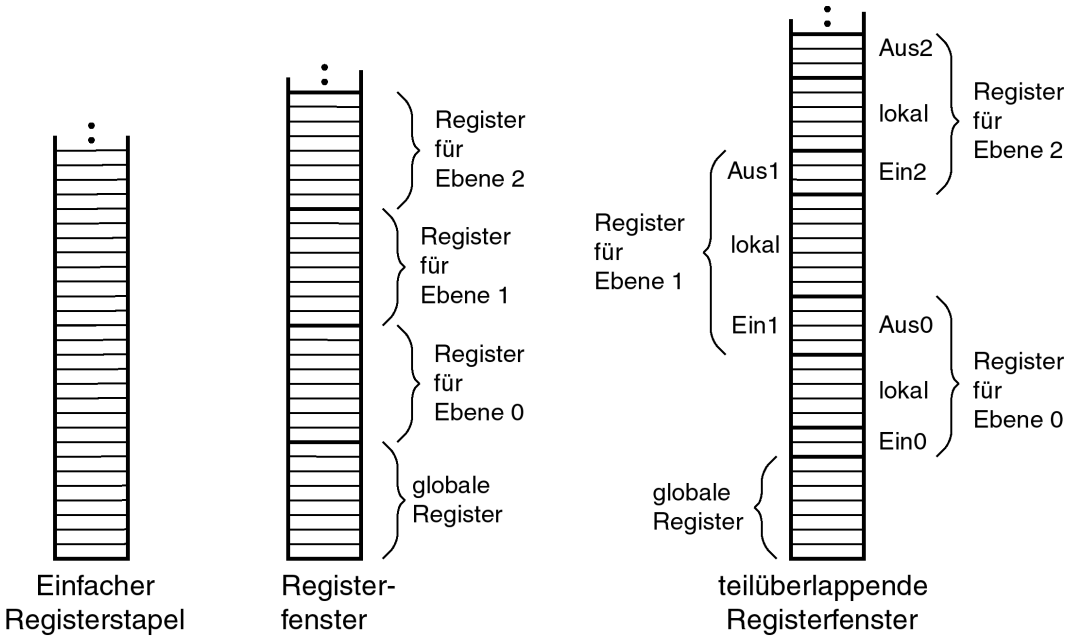


Figure 19.8: Verschiedene Organisationen der Register in einem RISC-Prozessor.

Hardware-Software-Schnittstelle (Instruction Set Architecture)

Die Hardware-Software-Schnittstelle, auch Instruction Set Architecture (*ISA*) (Befehlssatzschnittstelle) genannt, beschreibt die gesamte nach außen hin sichtbare Architektur:

- Die ISA umfasst den Befehlssatz, den Registersatz und das Speichermodell (Breite der Busse, die Größe und Beschaffenheit des Adressraumes und weitere Merkmale) [46].
- Die ISA ist maßgeblich für die die Erstellung von Maschinenprogrammen für diesen Prozessor durch Assembler und Compiler.
- Die ISA kann als Schnittstelle zwischen Software und Hardware betrachtet werden (Abb. 19.9).
- Die ISA ist mitentscheidend über Einsatzmöglichkeiten und Erfolg eines Prozessors.
- Oft wird eine ISA so gestaltet, dass sie sämtliche Elemente der ISA eines Vorgängers einschließt, man spricht von *Abwärtskompatibilität*.

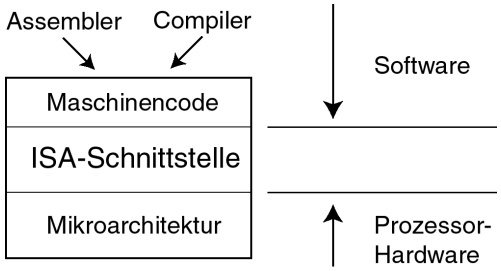


Figure 19.9:
Die Instruction Set Architecture (ISA) ist bei einem Mikroprozessor die Schnittstelle zwischen Software und Hardware.

Demgegenüber gehört die *Mikroarchitekturebene* nicht zur ISA.

- Die Mikroarchitektur umfasst alle internen Vorgänge des Prozessors, wie ALU-Betrieb, Mikroprogrammierung, Pipelining u.a.m.
- Die Mikroarchitektur garantiert in ihrer Gesamtheit das Funktionieren des Prozessors
- Um den Prozessor zu programmieren, braucht man die Mikroarchitekturebene nicht zu kennen

Bei vielen modernen Prozessoren ist die Trennung dieser beiden Ebenen nicht mehr so klar, wie sie einmal war. Ein Beispiel dafür sind die Pipelines der RISC-Prozessoren, die nicht mehr beliebig aufeinander folgende Maschinenbefehle verarbeiten können oder der SSE2-Befehlssatz der Pentium-Prozessoren, der eine explizite Steuerung der Caches zulässt. Um diese Prozessoren zu programmieren sind also doch Kenntnisse der Mikroarchitektur notwendig.

Interrupt-Controller

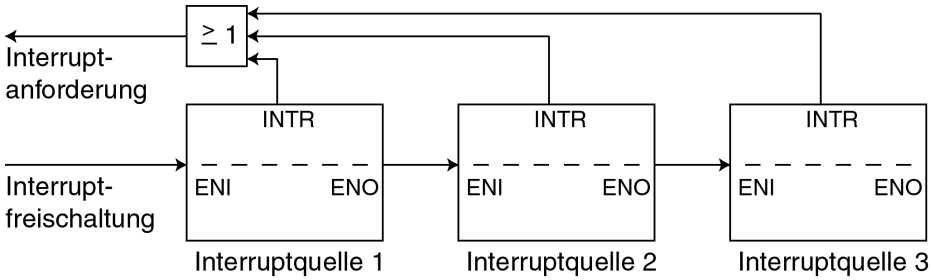


Figure 19.10: Daisy-Chaining von drei Interruptquellen. Interruptquelle 1 hat die höchste Priorität, Interruptquelle 3 die niedrigste. Nachteil: Die Prioritätenreihenfolge ist fest.

In PCs kommt ein Interrupt-Controller zum Einsatz (heute in den Prozessor integriert), der folgendes leistet:

- Alle Interruptsignale der Geräte werden an dem Interrupt-Controller angeschlossen
- Der Interruptcontroller gibt ein Signal an den Prozessor weiter
- Der Interruptcontroller ist programmierbar, d.h. in seine Register können passende Werte eingetragen werden.
- Er besitzt ein Maskenregister, über das die angeschlossenen Bausteine einzeln für Interrupts zugelassen oder gesperrt werden können.
- Er besitzt ein Prioritätsregister, darin werden die Prioritäten der Interrupts festgelegt.
- Beim *vektorisierten Interrupt* übermittelt er die Nummer des anstehenden Interrupts über den Datenbus an der Prozessor; das spart Zeit.

Zum Beispiel in Abb.19.11:
Interruptquellen 0, 1 und 7 fordern gleichzeitig einen Interrupt an. Im Maskenregister ist Interruptquelle 1 maskiert, es werden daher nur die Anforderungen der Interruptquellen 0 und 7 in das IRR übernommen. Interrupt 2 ist noch in

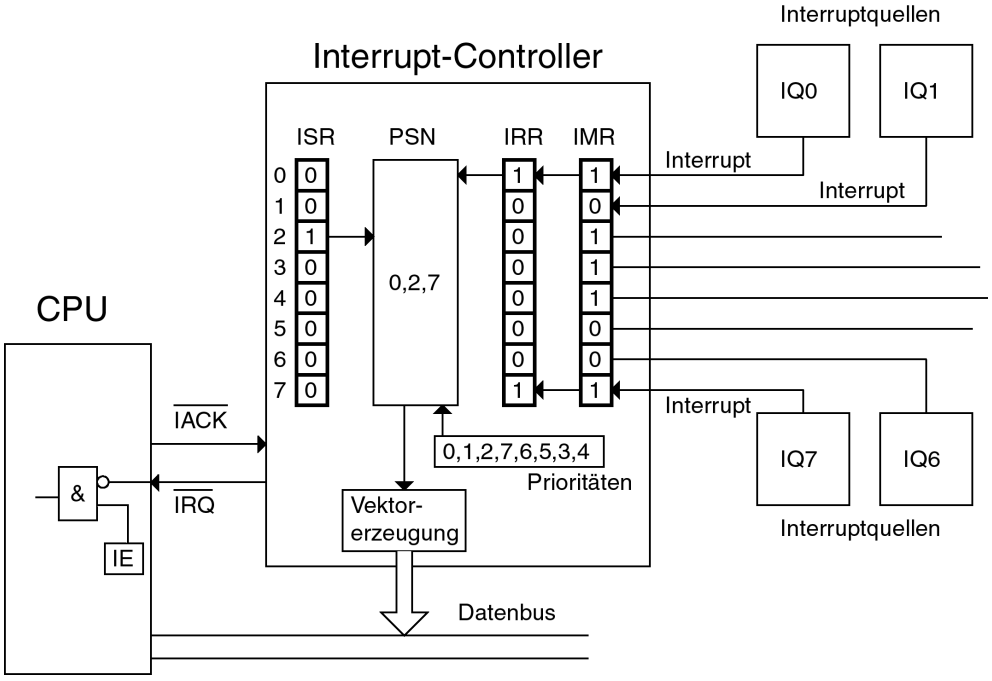


Figure 19.11: Ein typischer Interruptcontroller. (ähnlich 8259A)

Bearbeitung, das PSN muss also entscheiden. Auf Grund der einprogrammierten Prioritäten wird die Reihenfolge sein: 0,2,7. Die Bearbeitung von Interrupt 2 wird also unterbrochen. Es wird bei der CPU sofort ein Interrupt angefordert und der Interrupt-Controller übermittelt im folgenden Interrupt-Acknowledge-Zyklus die Nummer 0 über den Datenbus. Der Prozessor ruft jetzt die Behandlungsroutine für Interrupt 0 auf. Danach wird die unterbrochene Interrupt-Service-Routine von Interrupt 2 fortgesetzt. Nach deren Ende wird die Interruptanforderung 7 übermittelt und bearbeitet. Erst danach wird das bei Eintreten von Interrupt 2 laufende Programm fortgesetzt.

19.4 Ergänzung: Hilfsschaltungen

Zum Aufbau einfacher Systeme brauchen wir außer Mikroprozessor, Speicher und E/A-Bausteinen zwei Hilfsschaltungen: Taktgenerator und Einschaltverzögerung.

Taktgenerator

Er erzeugt ein systemweites Taktsignal, das in einem Mikroprozessorsystem alle Vorgänge synchronisiert. Für den Taktgenerator (auch astabile Kippstufe oder Multivibrator), gibt es verschiedene Schaltungsmöglichkeiten. Um die Frequenz stabil zu halten, wird dabei oft ein Schwingquarz eingesetzt.

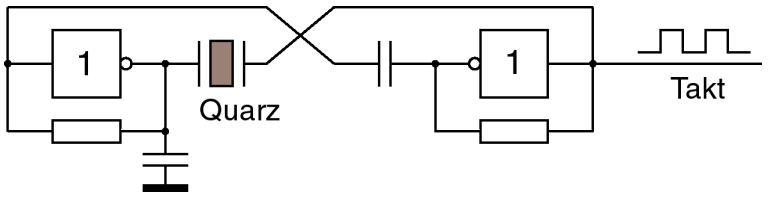


Figure 19.12: Ein Taktgenerator kann aus zwei Invertern aufgebaut werden. An beiden Inverter ist das Ausgangssignal an den Eingang zurückgeführt, dadurch ist die Schaltung prinzipiell instabil. Die Kondensatoren und Widerstände verlangsamen das ständige Umkippen. Der Quarz stabilisiert durch sein Resonanzverhalten die Taktfrequenz.

Einschaltverzögerung

- Arbeitet mit der Aufladezeit eines RC-Gliedes
- Bewirkt, dass das RESET-Signal erst deutlich nach dem Einschalten der Betriebsspannung deaktiviert wird.
- Die externen Schaltkreise des Systems (Speicher, E/A) können sich vor dem Bootvorgang stabilisieren
- Durch die Schmitt-Trigger werden die Signalflankensauber und steil geformt

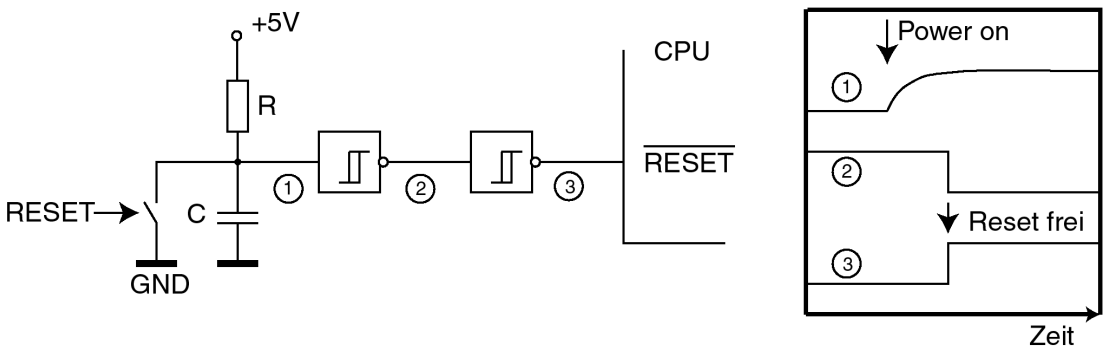


Figure 19.13: Typische Schaltung zur Einschaltverzögerung.

Übung: 19.1 Verständnisfrage

Was ist der Sinn einer Einschaltverzögerung?

Übung: 19.2 Frageblock

1. Welche Vor- und Nachteile hat die Harvard-Architektur gegenüber der von-Neumann-Architektur?
2. Warum kann auf ein Register schneller zugegriffen werden als auf eine Speicherzelle?
3. Skizzieren Sie, ähnlich wie auf Seite 170, die Abfolge der Steuersignale in einem Befehl, der ein Speicherwort in Register 1 kopiert. Die Speicheradresse soll dem Opcode als ein Operandenwort folgen.
4. Bestimmen Sie für die ALU 74181 die notwendigen Steuersignale für die folgenden Operationen:
   a) Die Addition zweier Operanden,
   b) die bitweise logische ODER-Verknüpfung zweier Operanden,
   c) die Invertierung des Operanden am A-Eingang.
5. Welche Adresse wird bei indizierter register-indirekter Adressierung angesprochen, wenn der Inhalt des Indexregisters 0020h ist, der Skalierungsfaktor auf 4 eingestellt ist und das Displacement 10h beträgt?
6. Welche Adresse wird bei nachindizierter speicher-indirekter Adressierung angesprochen, wenn der Inhalt des Speicheradressierungs-Registers 20h ist, das Displacement 1 gleich 1 ist, das Indexregister 1 enthält, der Skalierungsfaktor 1 ist und das Displacement 2 ebenfalls 1 ist? Der Inhalt des Speichers an den Adressen 20h – 23h sei:

60h	70h	80h	90h
20h	21h	22h	23h

7. Nennen Sie die Vorteile der Mikroprogrammierung.
8. Nennen Sie die Entwurfsziele von RISC-Prozessoren.
9. Was ist der Sinn einer Einschaltverzögerung?

20 Historie und Entwicklung der Mikroprozessortechnik

20.1 Geschichtliche Entwicklung der Mikroprozessortechnik

Die Geschichte der Computer verlief in vielen Generationen.

1. Generation: Mechanische Rechenmaschinen Rädertriebwerk, Addition und Subtraktion,

- Wilhelm Schickard, 1623
- Blaise Pascal, 1642
- Gottfried Wilhelm Leibniz, 1672
- Charles Babbage, 1833

fehleranfällig, Einzelstücke.

2. Generation: Rechner mit elektromagnetischen Relais

- Konrad Zuse, Z3, 1941
- Howard Aiken, MARK I, 1944

3. Generation: Rechner mit Elektronenröhren

- Atanasoff-Berry-Computer, 1938–1942, Lösen linearer Gleichungssysteme
- COLOSSUS, 1943–1946 Entschlüsselung des Enigma-Codes durch Briten
- ENIAC, 1946–1955, ballistische Tabellen für die US Armee
- IBM-Rechner bis 1958, Büroaufgaben

4. Generation: Transistor-Rechner

- Massachusetts Institute auf Technology (M.I.T.) TX-0, 1950
- Digital Equipment Corporation (DEC) PDP-1, 1961
- Control Data Corporation (CDC), CD6600, 1964

5. Generation: Rechner mit integrierten Schaltkreisen Nach der Erfindung des integrierten Schaltkreises durch Robert Noyce, 1958

- z.B. IBM System/360, 1964
- und andere

6. Generation: Rechner mit Mikroprozessoren Nach der Erfindung des Mikroprozessors durch Intel 1971

- Intel 4004, 1971

- Intel 8008, 1972
- Motorola 6800, 1974
- und viele andere bis heute

Die Computer- und speziell die Mikroprozessortechnik ist ein Gebiet, das sich rasant entwickelt:

Die Komplexität Der erste Mikroprozessor (i4004) hatte ca. 2300 Transistoren, heute werden Prozessoren mit mehr als 1 Milliarde Transistoren gefertigt

Der Integrationsgrad Die Anzahl von Elementarschaltungen (z.B. Gatter) ist von weniger als 100 (SSI) auf mehr 1 Million pro Chip angestiegen.

Der Arbeitstakt Die ersten Mikroprozessoren wurden mit Taktfrequenzen unterhalb von 1 MHz betrieben, aktuelle PC-Prozessoren laufen mit über 4 GHz

Die Verarbeitungsbreite Der i4004 arbeitete mit 4 Bit Verarbeitungsbreite, heute sind 64 Bit Verarbeitungsbreite üblich.

Ein anderer Vergleich: Die Rechner der Apollo-Missionen (Mondlandung 1969), CPU, die aus 5000 NOR-Gattern, Arbeitsspeicher 2048 Byte, 36 kWorte ROM. Das Programm im "'core rope ROM"' (Jeder Speicherplatz ein kleiner Ringkern, Fädeltechnik)

20.2 Die schnelle Entwicklung der Mikroprozessortechnik – das Mooresche Gesetz

> Die Anzahl der Transistoren in den integrierten Schaltungen verdoppelt sich regelmäßig in 18 Monaten. *(Moores Gesetz)*

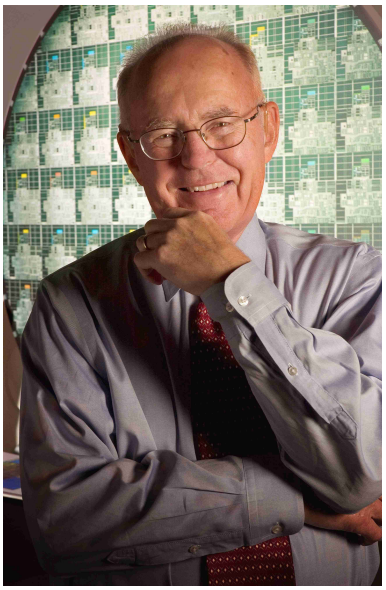


Figure 20.1:
Gordon Moore, einer der Mitbegründer von Intel, hat das Gesetz von der Zunahme der Komplexität der ICs formuliert. (Quelle: www.intel.com)

Ohne die rasante Entwicklung nach dem Mooreschen Gesetz wäre die moderne Mikroelektronik nicht möglich!

Figure 20.2: Das erste Handy, Motorola DynaTac8000x; 4000$, 800g, nur telefonieren, 1h Gesprächszeit (Quelle: Wikipedia)

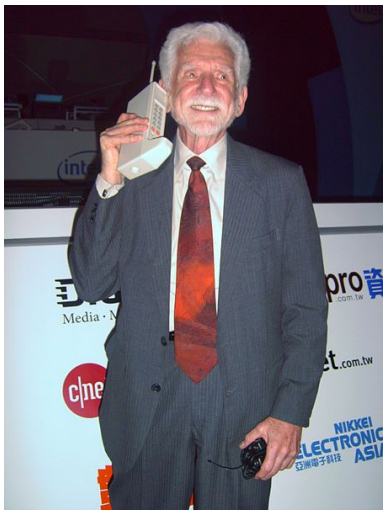


Figure 20.3: der Entwickler des ersten Handys Dr. Martin Cooper. (Quelle: Wikimedia)

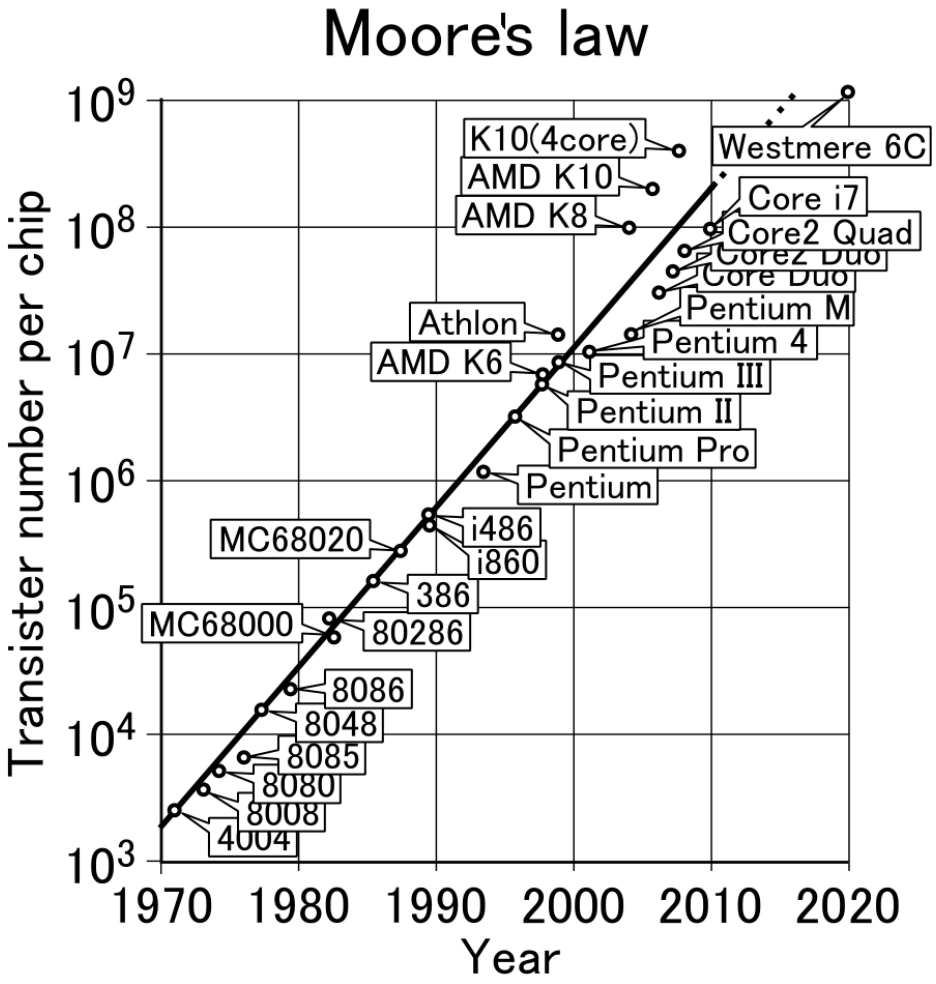


Figure 20.4: Mikroprozessoren entwickeln sich nach Moores Gesetz. (Quelle: Wikimedia)

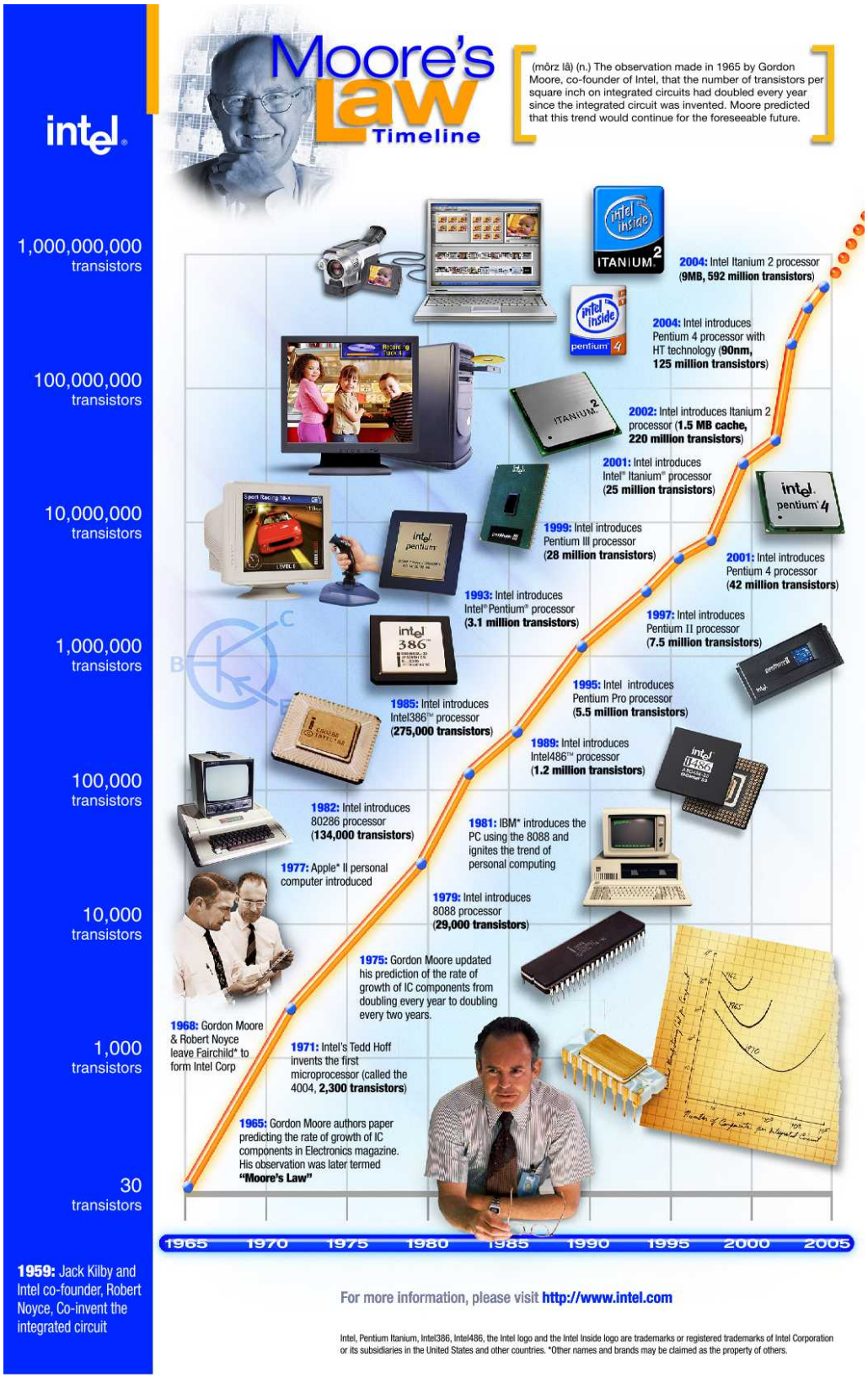


Figure 20.5: Die technische Entwicklung mit dem Mooreschen Gesetz. (Quelle: www.intel.com)

Übung: 20.1 Testfrage: Mooresches Gesetz

1. Nennen Sie die bisherigen Rechnergenerationen und ihren ungefähren Einführungszeitpunkt.
2. Die kleinste Ausdehnung der Bauelemente eines integrierten Schaltkreises (Strukturbreite) aus dem Jahr 2003 beträgt 90 nm. Welche Strukturbreite erwarten Sie im Jahr 2015, welche im Jahr 2027?

Bibliography

[1] ACPI *Advanced Configuration and Power Interface Specification* Version 4.0a, April 2010, www.acpi.com

[2] Altenburg, J. , Bögeholz, Harald: *Mikrocontroller-Praxis* Teil 1–4, c't Magazin 24/2003, 25/2003, 5/2004, 6/2004

[3] ARM Ltd: *Cortex-M3* Technical Reference Manual, Revision: r1p1 www.arm.com

[4] Bähring, H.: *Mikrorechner-Technik*, Bd.1. Mikroprozessoren und Digitale Signalprozessoren, Bd.2. Busse, Speicher, Peripherie und Mikrocontroller, Springer–Verlag, 3. Aufl., Berlin, 2002

[5] Baetke, F.: *IA-64: Strickmuster für den Computer der Zukunft*, Spektrum der Wissenschaft, Rechnerarchitekturen, Dossier 4/2000, S.74

[6] Barrett, S.F., Pack, D.J. *Embedded Systems*, Design and Applications with the 68HC12 and HCS12, Pearson Education, 2005

[7] Bauer, F.L.: *Informatik*, Führer durch die Ausstellung, Deutsches Museum, München, 2004

[8] Beierlein, Th. und Hagenbruch, O.: *Taschenbuch Mikroprozessortechnik*, Fachbuchverlag Leipzig, München, Wien, 1999

[9] Benz, B.: *Spannungsfeld, Prozessoren: Sparsamkeit kontra Stabilität und Taktfrequenz*, c't Magazin 17/2010, S.166

[10] Benz, B.: *Nachbrenner – Prozessor-Turbos von AMD und Intel*, c't Magazin 16/2010, S.170

[11] Benz, B.: *Phenom inside – AMDs Vierkernprozessoren im Detail*, c't Magazin 2/2008, S.80

[12] Beuth, K.: *Elektronik 2 – Bauelemente*, Vogel Buchverlag, 15.Aufl., Würzburg, 1997

[13] Beuth, K.: *Elektronik 4 – Digitaltechnik*, Vogel Fachbuch Verlag, 9.Aufl., Würzburg, 1992

[14] Bleul, A.: *Computer ad astra*, c't Magazin 5/1999, S.108

[15] Brinkschulte, U., Ungerer, T.: *Mikrocontroller und Mikroprozessoren*, Springer, Berlin 2002

[16] Elektronikpraxis: *Energieeffizienz und Eco-Design, Sonderheft*, Würzburg, April 2008

[17] El-Sharkawy, M.: *Digital Signal Processing Applications with Motorola's DSP56002 Processor*, Prentice Hall, London, 1996

[18] Flik, Th.: *Mikroprozessortechnik*, Springer-Verlag, 7. Aufl., Berlin, 2005

[19] König, P.: *Sparprogramm - Am Rechner Geldbeutel und Umwelt schonen*, c't Magazin 4/2008, S.78

[20] Freescale Inc.: *Technical Data DSP56F801 16-bit Digital Signal Processor, Rev. 15, 10/2005*

[21] Freescale Inc.: *DSP56800 16-Bit Digital Signal Processor, Family Manual, Rev. 3.1, 11/2005*

[22] Hennessy, J.L. und Patterson, D.A.: *Rechnerarchitektur* Vieweg-Verlag, Braunschweig/Wiesbaden 1994.

[23] Herrmann, P.: *Rechnerarchitektur*, Vieweg-Verlag, 3. Aufl., Braunschweig/Wiesbaden 2002

[24] Infineon Technologies AG,: *H*albleiter , Publicis MCD Corporate Publishing, 2.Aufl., Erlangen und München, 2001

[25] Infineon Technologies: *C167CR/SR Derivatives, 16-Bit Single-Chip Microcontroller* Data Sheet V3.3, 2.2005, User's Manual V3.2, 5.2003, www.infineon.com

[26] Intel Corporation: *Intel® 64 and IA-32 Architectures Software Developer's Manual* Volume 1: Basic Architecture, 2010, Volume 2,3 : Instruction Set Reference, 2010, Volume 4,5: System Programming Guide, 2010, alle: www.intel.com

[27] Intel Corporation: *Intel Itanium Architecture* Software Developer's Manual Volume 1: Application Architecture, Rev. 2.2, 2006, Volume 2: System Architecture, Rev. 2.2, 2006, Volume 3: Instruction Set Reference, Rev. 2.2, 2006, alle: www.intel.com

[28] Johannis, R.: *Handbuch des 80C166*, Feger+Co. Hardware+Software Verlags OHG, Traunstein, 1993

[29] Koopman, P.: *Microcoded Versus Hard-wired Control*, BYTE 2987, Jan. 1987, S.235

[30] Kopp, C.: *Moore's law and its implication for information warfare*, 3rd International AOC EW Conference, Jan. 2002

[31] Lindner, H.: Brauer, H. und Lehmann, C., *Taschenbuch der Elektrotechnik und Elektronik*, Fachbuchverlag Leipzig, 8. Aufl., München, Wien, 2004

[32] Malone, S.M.: *Der Mikroprozessor, eine ungewöhnliche Biographie*, Springer-Verlag, Berlin, Heidelberg, 1996

[33] Mengel, St., Henkel, J.: *Einer speichert alles, MRAM – der lange Weg zum Universalspeicher*, c't Magazin 18/2001, S.170

[34] Messmer, H.P.: *Das PC-Hardwarebuch*, Addison-Wesley Deutschland, 6. Aufl., München, 2000

[35] Mildenberger, O.: *System- und Signaltheorie* Vieweg-Verlag, Braunschweig/Wiesbaden 1995.

[36] Müller, H. und Walz, L.: *Elektronik 5 – Mikroprozessortechnik*, Vogel Buchver-

lag, Würzburg, 2005

[37] Nus, P.: *Praxis der digitalen Signalverarbeitung mit dem DSP-Prozessor 56002*, Elektor-Verlag, Aachen, 2000

[38] Rohde, J.: *Assembler GE-PACKT*, mitp-Verlag, Bonn, 2001

[39] Schmitt, G.: *Mikrocomputertechnik mit dem Controller C 167* , Oldenbourg-Verlag, 2000

[40] Stiller, A.: *Architektur für echte Programmierer, IA-64, EPIC und Itanium* c't Magazin 13/2001, S.148

[41] Stiller, A.: *Die Säulen des Nehalem – Die Core-Architektur des neuen Intel-Prozessors*, c't Magazin 25/2008, S.174

[42] Stiller, A.: *Mikronesische Bauwerke – Architektur und Performance der Netbook-Prozessoren*, c't Magazin 18/2008, S.96

[43] Rauber, T. und Rünger, G.: *Parallele Programmierung* eXamen-press 2012

[44] Sturm, M.: *Mikrocontrollertechnik* am Beispiel der MSP430-Familie, Hanser-Verlag 2006

[45] Simon, D.E.: *An Embedded Software Primer* Pearson, 1999 15.1, 15.4

[46] Tanenbaum, A.S. und Goodman, J.: *Computerarchitektur*, 5. Aufl., Pearson Studium, München, 2006 19.3

[47] Texas Instruments: *MSP430x22x2, MSP430x22x4 Mixed Signal Controller datasheet*: www.ti.com 14.2

[48] Texas Instruments: *MSP430x2xx Family User's Guide* www.ti.com 4.2

[49] Windeck, C.: *Spar-O-Matic, Stromsparfunktionen moderner x86-Prozessoren*, c't Magazin 15/2007, S.200

[50] Werner, M.: *Signale und Systeme* Vieweg-Verlag, 2. Aufl., Braunschweig/Wiesbaden 2005.

[51] Wittgruber, F.: *Digitale Schnittstellen und Bussysteme*, 2. Aufl., Vieweg-Verlag, Braunschweig/Wiesbaden, 2002

[52] Wüst, K.: *Mikroprozessortechnik* Vieweg+Teubner Verlag, 4. Aufl., Wiesbaden 2010.

[53] Yiu, J., *The definitive guide to the ARM Cortex-M3*, Newnes, 2. Ed.l., Amsterdam, 2010