Hochschule Esslingen
University of Applied Sciences

Echtzeitsysteme

Prof. Dr. (Purdue Univ.) Jörg Friedrich
Fakultät Informationstechnik
Hochschule Esslingen

Copyright 2007 Hochschule Esslingen
September 2007

Inhalt

1

Einführung

1.1 Organisatorisches

1.1.1 Vorlesungszeiten und Räume

- Mi, 9:30 bis 12:45 in Raum H7
- Zwei offizielle Labortermine pro Gruppe in Raum F1.301, werden noch bekannt gegeben

1.1.2 Unterlagen und Sprechzeiten

- Vorlesungsmaterial befindet sich auf dem T-Laufwerk
- Sprechzeiten: Di, 9:00 bis 10:00
- E-Mail: joerg.friedrich@hs-esslingen.de

1.1.3 Prüfungsmodalitäten

- Für den Diplomstudiengang (TI7) findet die Prüfung zusammen mit der Prüfung „Bussysteme“ statt. Dauer: 150 Minuten. Gewichtung: 100 Minuten PDV, 50 Minuten Bussysteme.
- Für den Bachelorstudiengang 90-minütige Prüfung

1.1.4 Hinweise zu den Übungen und zum Labor

- Bei den Laboren geht es um die Programmierung von Echtzeitsystemen. Das kann man nicht durch Ausfüllen von Lückentext oder in wenigen Stunden lernen.
- Die Labore erfordern eine aufwändige Vorbereitung (ca. 10 bis 20 Stunden pro Labor, je nach Erfahrung und Vorkenntnissen).
- Sie dürfen sich über einen zu hohen Aufwand für die Labore beschweren. Ich gehe von einem durchschnittlichen Arbeitsaufwand von 1,5 Zeitstunden pro Woche für jede Semesterwochenstunde aus. Wenn Sie die überschreiten, melden Sie sich.
- Zu den Laborterminen sollten Sie im wesentlichen mit den Aufgaben fertig sein. Die Zeit dort reicht nicht aus, die Aufgaben zu bearbeiten. Sie werden dort besprochen.
- Sie haben jederzeit Zugang zum Labor, können einen großen Teil der Aufgaben aber schon auf Ihrem eigenen Rechner bearbeiten.
- Jeder in einer Laborgruppe muss alle Fragen beantworten können und selbst zur Lösung aktiv (nicht nur durch Zuschauen) beigetragen haben.
- Erster Ansprechpartner bei Fragen und Problemen im Labor ist Herr Trybek, Raum F1.407, Tel.
- Ich stehe Ihnen gerne im Rahmen meiner Möglichkeiten bei Problemen zur Verfügung.

1.2 Voraussetzungen und Struktur der Vorlesung

- Thema dieser Vorlesung sind **Echtzeitsysteme und deren Programmierung**.
- Vorausgesetzt werden Kenntnisse in C-Programmierung, Elektronik und Digitaltechnik
- Gliederung:

1. Einführung, Begriffe, Softwareentwicklung für Echtzeitsysteme
2. Kopplung technischer Prozesse an Rechner, Peripherie, Interrupts
3. Modellierung von Echtzeitsystemen mit UML
4. Einführung Echtzeitbetriebssysteme
5. Task-Verwaltung
6. Task-Synchronisation und -Kommunikation
7. Ablaufsteuerung
8. Timing Services
9. Memory Management in Echtzeitsystemen
10. POSIX.4, VxWorks, QNX

1.3 Literaturhinweise

1. Textbuch zur Vorlesung: Echtzeitsysteme, H. Wörn, U. Brinkschulte, Springer, ISBN 3-540-20588-8, € 39,95
2. Real Time Concepts for Embedded Systems, Q. Li, CMP Books, ISBN 1-57820-124-1, €42,-
3. Modern Operating Systems 2nd Ed., A. Tanenbaum, Prentice Hall, ISBN 0-13-092641-8
4. Real-Time Systems, J. Liu, Prentice Hall, ISBN 0-13-099651-3
5. MicroC/OS-II, The Real Time Kernel, J. Labrosse, CMP Books, ISBN 1-57820-103-9
6. OSEK, M. Homann, mitp-Verlag, ISBN 3-8266-1552-2
7. Scheduling in Real-Time Systems, F. Cottet et al., Wiley, ISBN 0-470-84766-1
8. Webseite: http://www.embedded.com
9. Webseite: http://www.osek-vdx.org
10. Webseite: http://www.dspace.de
11. Webseite: http://vector-informatik.de
12. Webseite: http://www.etas.de
13. Webseite: http://www.automation.siemens.com
14. Webseite: http://www.windriver.com
15. Webseite: http://www.qnx.com
16. Webseite: http://www.linux-automation.de

1.4 Sonstige Unterlagen

- Dokumentation zum verwendeten Freescale-Rechner (auf Studi-CD)
- Dokumentation zu RMOS3 (auf Studi-CD)
- OSEK-Spezifikationen (auf Studi-CD)

1.5 Einordnung der Vorlesung

Echtzeitsysteme
Steuern ereignisdiskreter Systeme und Softwarearchitekturen unter Echtzeitbedingungen

Computerarchitektur 3
Schnittstelle zwischen Rechnerhardware und Software, Assembler, Peripheriebausteine, Rechnerleistung

Systemtechnik
Simulieren, Steuern und Regeln kontinuierlicher Systeme

Computerarchitektur 2
Hardware-Konzepte und Baugruppen eines Rechners, VHDL-Programmierung

Informatik
Softwarentwurf, Programmieren, Betriebssysteme, Rechnerorganisation

Computerarchitektur 1
Grundbausteine der Digitaltechnik, Logik- und Registerschaltungen

Physik
Elektrotechnik
Signale und Systeme unter Echtzeitbedingungensysteme

1.6 Eingebettete Systeme, dedizierte Systeme und Echtzeitsysteme

Versuch einer Definition und Abgrenzung

- **Eingebettete Systeme**: werden durch Rechner gesteuert, Teil eines umfangreicheren Gesamtsystems, keine PC-üblichen Benutzerschnittstellen.
- Beispiele: Airbag-Steuergerät
  ABS
  Mikrowelle, Waschmaschine
  Roboter-Steuerung
  Personal Digital Assistant
  Mobiltelefon
- **Dedizierte Systeme**: Systeme, die für einen (meist wirklich nur genau einen) Zweck gebaut werden
- Beispiele: siehe oben, bis auf Personal Digital Assistant (Benutzer kann Software aufspielen nach seinem Wunsch)
- **Echtzeitsysteme**: meist dedizierte System, die für korrekte Funktion Zeitbedingungen einhalten müssen. Z.B. Airbag-Fehlfunktion, wenn zu spät gezündet wird. Nennt man auch *Realzeitsysteme*.

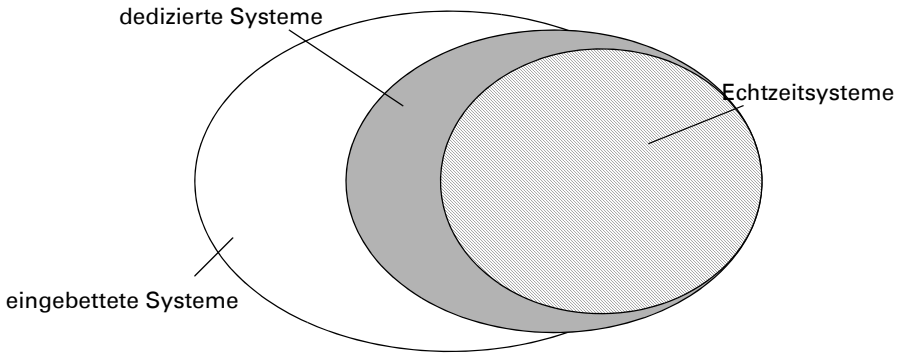


Bild 1.1: Abgrenzung eingebettete, dedizierte und Echtzeitsysteme

Echtzeitsysteme: *Korrektheit* der Ergebnisse genauso wichtig wie Erfüllung *der Zeitbedingungen.*
Definition nach DIN 44300:

> Echtzeit- bzw. Realzeitbetrieb eines Rechners, wenn Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die anfallenden Daten können je nach Anwendungsfall nach einer zufälligen zeitlichen Verteilung oder zu bestimmten Zeitpunkten auftreten.

Echtzeitsysteme stellen Anforderungen an

- *Rechtzeitigkeit*
- *Gleichzeitigkeit*
- *zeitgerechte Reaktion auf spontane Ereignisse*

Echzeitsysteme bestehen aus Hardware- und Softwarekomponenten. Diese *erfassen* und *verarbeiten* anfallende *interne* (aus dem Echtzeitsystem selbst kommende) und *externe* (aus der Umgebung kommende) *Daten* und *Ereignisse*.
Wir unterscheiden zwischen

- *asynchron* arbeitenden Echtzeitsystemen
- *synchron* bzw. *zyklisch* arbeitenden Echtzeitsystemen

Wir unterscheiden zwischen

- Echtzeitsystemen in Massenprodukten (*Produktautomatisierung*)
- Echtzeitsystemen in technischen Großanlagen (*Prozessautomatisierung*)

Anforderungen an dedizierte Systeme sind:

- *Niedrige Herstellungskosten*. Schränken möglichen Ressourcen-Verbrauch stark ein. Erfordern Hardware-Software-Codesign.
- *Niedriger Energieverbrauch*. Bei mobilen Geräten sehr wichtiges Entwurfs-Kriterium. Bestimmt Kosten mit (z.B. für Entwärmung).
- *Hohe Zuverlässigkeit* (engl. *reliability*). Betrifft Hard- und Software. Fehler können Schaden für Leib und Leben bedeuten. Produkthaftung nicht über Lizenzbedingungen einschränkbar. Manche Geräte nur schwer zugänglich (z.B. Richtfunkgerät auf der Zugspitze). Möglicher Imageschaden für Gesamtprodukt und Marke (Beispiel: Daimler-Chrysler).
- *Hohe Verfügbarkeit*. Systeme müssen rund um die Uhr laufen (schließt z.B. Reorganisation wie Garbage-Kollektion aus).

1.7 Problemstellung und Anforderungen

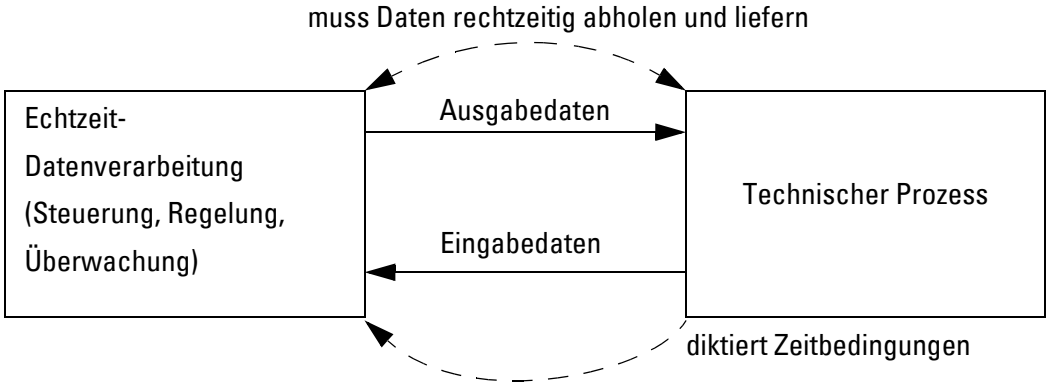
- Nicht-Echtzeitsysteme: *logische Korrektheit* bedeutet Korrektheit
- Echtzeitsysteme: *logische Korrektheit plus zeitliche Korrektheit* bedeutet Korrektheit

1.7.1 Rechtzeitigkeit

Bedeutung: Ausgabedaten müssen *rechtzeitig* zur Verfügung gestellt werden.

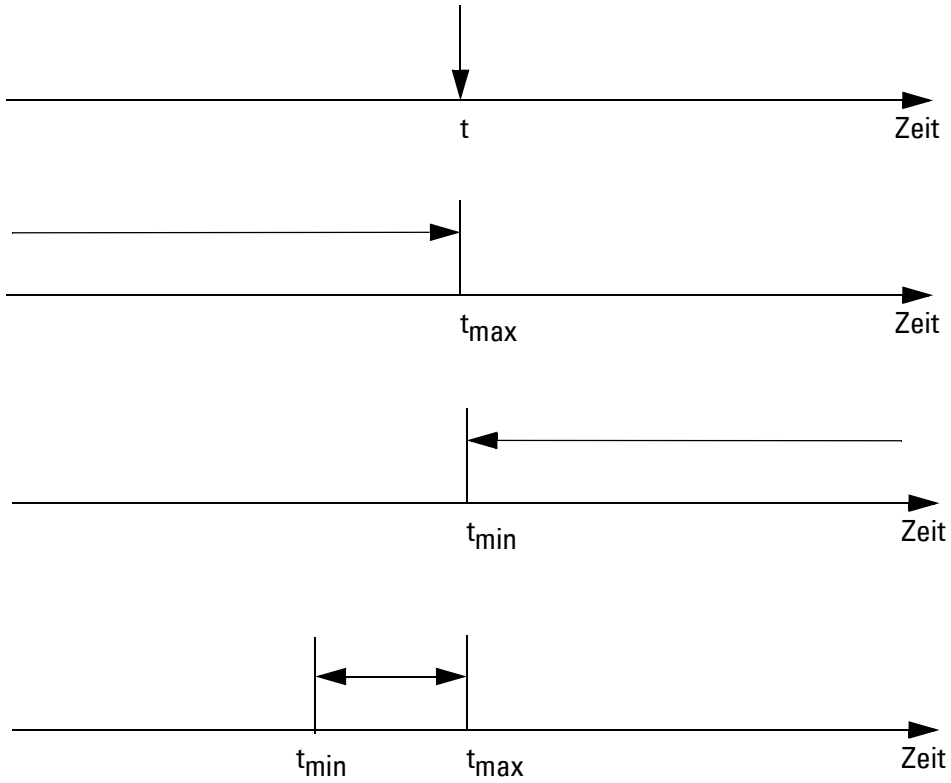
Erfordert indirekt auch die rechtzeitige Abholung von Eingangsdaten.

Technischer Prozess *diktiert Zeitbedingungen*.



Verschiedene Formen der Zeitbedingungen:

- Angabe eines ***genauen Zeitpunktes*** (Aktion muss genau zu diesem t stattfinden); ***Beispiel: Stoppen eines Fahrzeugs an einem genauen Punkt***
- Angabe eines ***spätesten Zeitpunktes*** (Zeitschranke, ***Deadline***)
- Angabe eines frühesten Zeitpunktes ***(z.B. Entladen nicht vor Beladen, wann spätestens ist egal)***
- Angabe eines Zeitintervalls (Aktion muss innerhalb dieses Zeitintervalls durchgeführt werden)



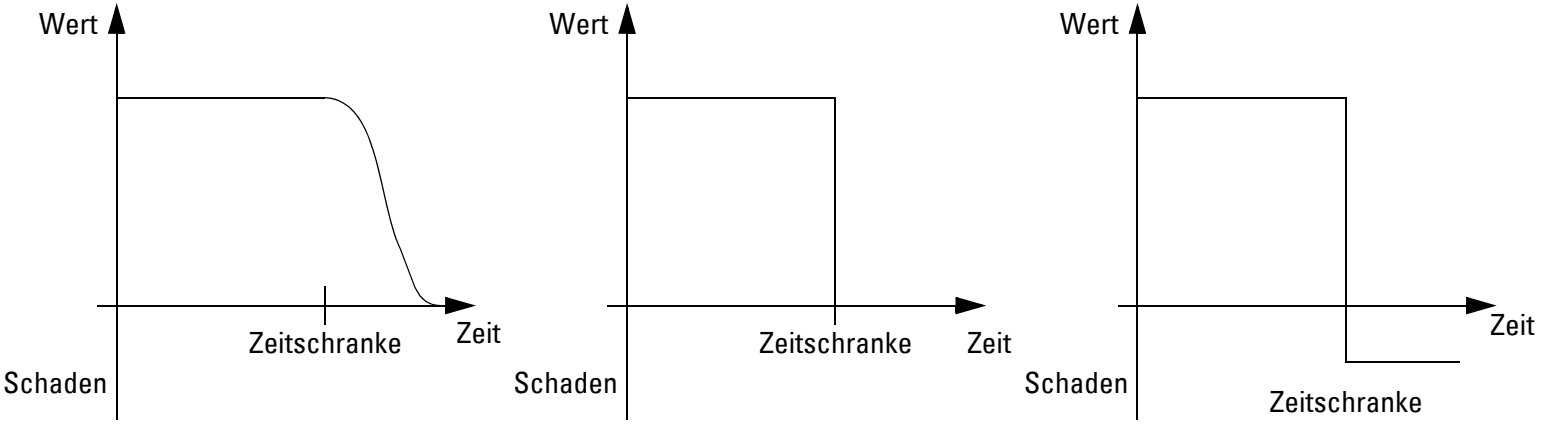
Unterteilung der *Zeitbedingungen* in

- *periodische* Zeitbedingungen
- *aperiodische* Zeitbedingungen

Weitere Unterteilung der *Zeitbedingungen* in

- *absolute* Zeitbedingungen
- *relative* Zeitbedingungen

Definitionen von harten, festen und weichen Echtzeitbedingungen:



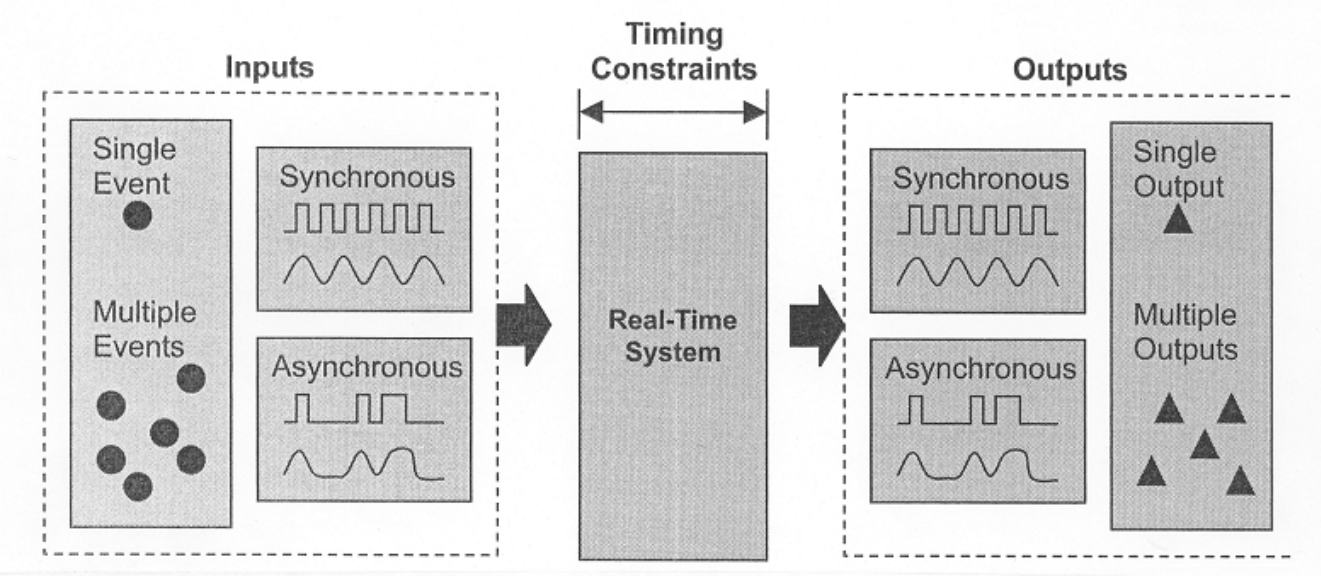
weich *fest* *hart*

Drei verschiedene Ansätze für Kriterien:

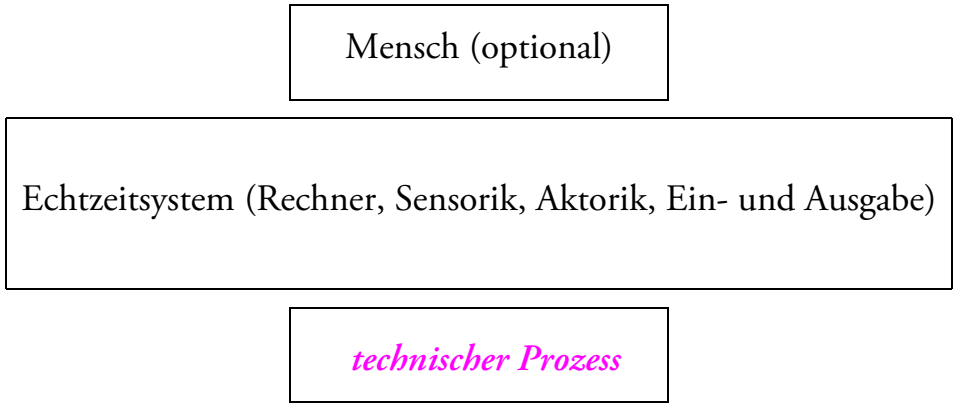
- nach Kritikalität (entsteht Schaden oder nicht)*; Beispiel: Airbag, Videoprozessor*
- nach Nützlichkeit (wie nützlich sind zu späte Ergebnisse)*; Problem hier: die Nützlichkeitsfunktion*
- nach zulässiger Häufigkeit von Zuspätkommen*; z.B. muss in 99,999% aller Fälle Zeitschranke einhalten*

1.8 Modell eines Echtzeitsystems

Ein einfaches Modell:



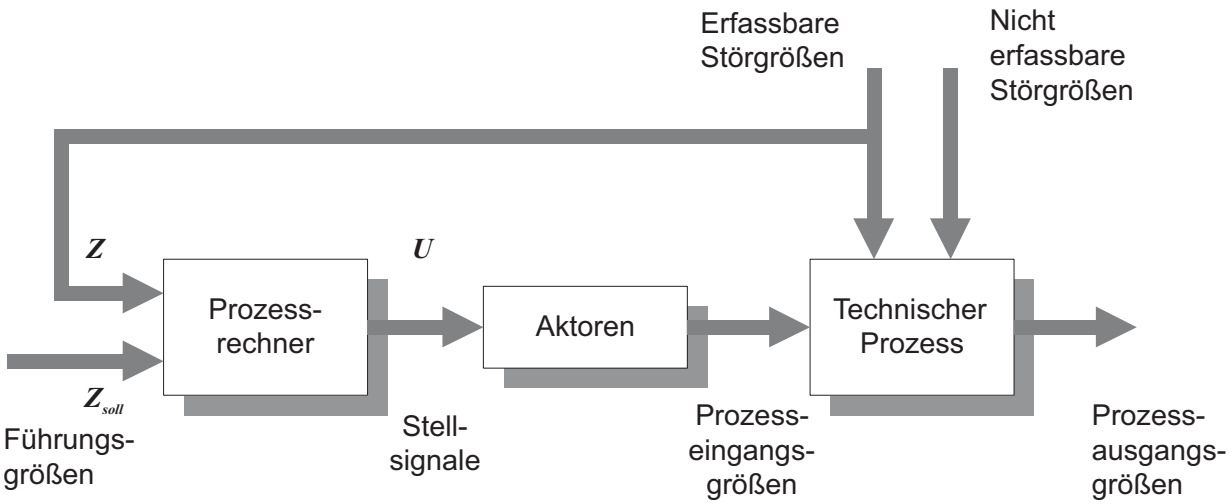
Andere Sicht:



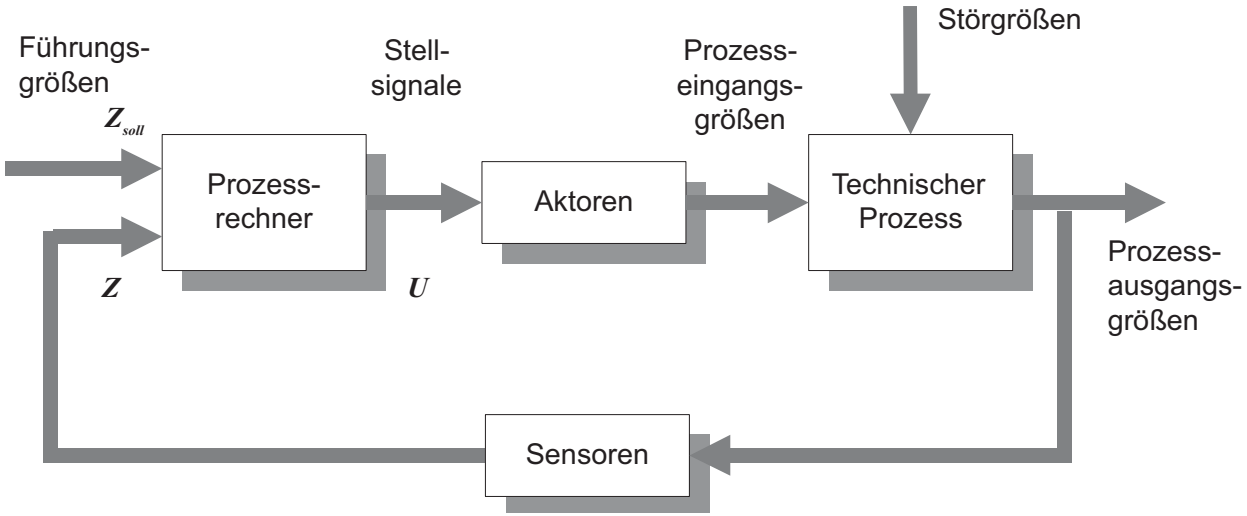
Echtzeitsysteme werden benutzt, um technische Prozesse zu

- *überwachen*: Zustand des technischen Prozesses über Sensoren erfassen und dem Nutzer melden. kritische Zustande selbständig handhaben, z.B. durch Abschalten, Einschalten
- *steuern:* Echtzeitsystem erzeugt *Führungsgrößen* und beeinflusst damit Steuer- und Stellglieder (*Aktoren*) nach vorgegebenem Algorithmus oder gemäß vorgegebenem Ziel
- *regeln:* Echtzeitsystem erzeugt Führungsgrößen, um die *Istwerte* vorgegebenen *Sollwerten* anzupassen. Dazu ist eine *Rückkopplung* von Istwerten die von *Sensoren* aufgenommen werden, auf das Echtzeitsystem notwendig.

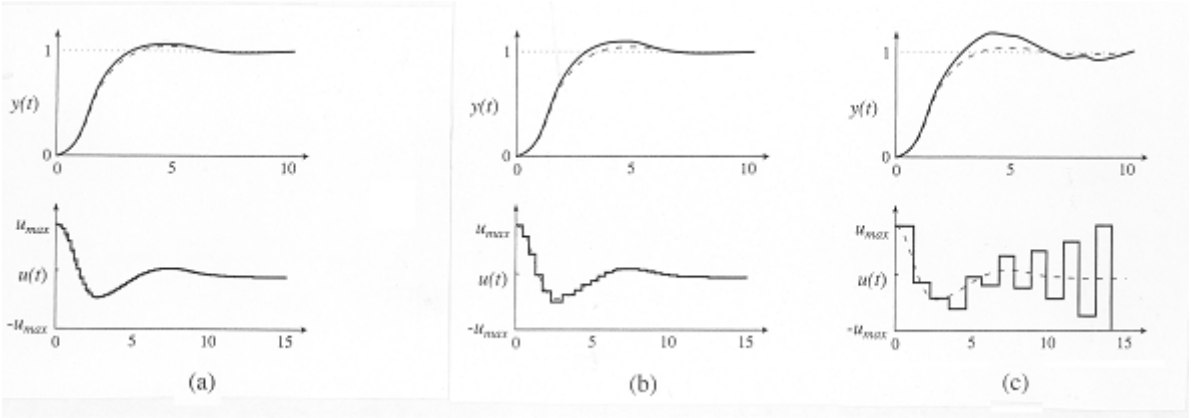
Prinzip der *Steuerung*:



Prinzip der *Regelung*:



Echtzeitsysteme müssen mit *Sensoren* und *Aktoren* an den technischen Prozess *gekoppelt* werden.
Das Zeitverhalten von Echtzeitsystemen hat Auswirkungen auf das Regelverhalten:



1.9 Klassifikation technischer Prozesse

Nach DIN 6620 ist ein technischer Prozess so definiert:

ein Prozess ist die Umformung und / oder der Transport von Materie, Energie und / oder Information.

Wir unterscheiden

- *diskrete* Prozesse, z.B. Verkehrsampel, Teilefertigung, Lagerhaltung, Airbag-Steuerung. Unterscheidbare Prozesszustände folgen aufeinander. Übergang wird durch Ereignisse augelöst.
- *stetige* Prozesse, z.B. chemische Prozesse, digitale Signalverarbeitung
- *hybride* Prozesse, z.B. Verbrennungsvorgang im Motor, Festo-Anlage im Labor

Das *Prozessmodell* ist ein abstraktes Abbild des realen Prozesses. Es beschreibt den Ausgang des Systems bei gegebenen Eingangsgrößen. Oft sind auch *Störgrößen* zu berücksichtigen.
Man unterscheidet zwischen *kontinuierlichen* und *diskreten* Systemmodellen. Ändert sich das Systemmodell über der Zeit, spricht man von einem *dynamischen* System, sonst von einem *statischen*.
Die Modellierung von kontinuierlichen Systemen ist Gegenstand der *Regelungstechnik*.

1.10 Typen von Prozesssteuerungen

In der Praxis finden wir fünf Typen von Prozesssteuerungen:

- automatisierte Produkte mit eingebautem Mikrorechner (benutzen wir im Labor)
- Steuerungen mit Industrie-PCs (IPC) (benutzen wir im Labor)
- *Speicherprogrammierbare Steuerungen (SPS)*
- Numerische Steuerungen, z.B. für Werkzeugmaschinen (NC)

Roboter-Steuerungen (RC) (haben wir im Labor, benutzen wir aber nicht)

2

Softwareentwicklung für Echtzeitsysteme

2.1 Labor- und Übungsumgebung

In dieser Vorlesung benutzen wir als Beispiel- und Übungsplattform zwei Systeme:

- ein Entwicklungsboard mit einem 16-Bit Mikrocontroller vom Typ Freescale 68HCS12 und verschiedenen Peripheriekomponenten (Dragon12-Board der Firma EVBPlus) und einem kleinen OSEK-ähnlichen Betriebssystem. Die sehr gut ausgestattete Boardhardware kostet ca. 130 Euro. Die gleiche Umgebung wird in den Vorlesungen „Computerarchitektur 3“ und „Systemtechnik“ verwendet.
- ein Industrie-PC-System aus der Industrieautomatisierung (SICOMP, RMOS3 der Firma Siemens).

Die Entwicklungsumgebung zum Dragon12-Board ist kostenlos erhältlich (Studi-CD).

Speicheradressen und Dateninhalte werden in dieser Vorlesung in hexadezimaler Notation wie in der Programmiersprache C angegeben. Der hexadezimale Wert 0xFE entspricht z.B. dem dezimalen Wert 254.

2.2 Entwicklungs- und Zielumgebung

Ressourcen dedizierter Systeme sind

- aus Kostengründen in der Regel *knapp bemessen*
- auf die jeweilige Anwendung zugeschnitten (z.B. keine Standard-Benutzerschnittstellen wie Tastatur und Bildschirm).

2.2.1 Entwicklungswerkzeuge

Für die Entwicklung der Software werden deshalb zwei Systeme benötigt:

- ein *Entwicklungsrechner* („*host*“)
- ein *Zielsystem*, das dedizierte System („*target*“)

Als Entwicklungsplattform dienen heutzutage fast ausschließlich Desktop-Systeme mit Windows oder Unix-Betriebssystem.
Auf der Entwicklungsplattform benötigt man mindestens

- einen *Editor*
- einen *Cross-Compiler*
- einen *Linker* und evtl. Locator (sind manchmal miteinander integriert, z.B. beim CodeWarrior)

Cross-Compiler: Compiler, der auf Entwicklungsplattform läuft, aber Maschinencode für Zielsystem erstellt. Zum Beispiel läuft die Codewarrior-Entwicklungsumgebung für das Labor auf einem PC und erzeugt Code für den 68HCS12-Mikrocontroller des Dragon12-Boards.
Für *jeden Mikroprozessortyp* im Zielsystem benötigt man dazu *passende Cross-Compiler und Linker*.
IDE: integrierte Entwicklungsumgebungen an (Integrated Development Environments). Beinhaltet die oben erwähnten Werkzeuge und weitere wie z.B. einen Debugger oder Systeme zum Bauen und Verwalten der Softwareprodukte.

2.2.2 Schnittstellen

Code muss vom Entwicklungssystem auf Zielsystem gebracht werden: *Schnittstelle erforderlich.*
Übliche Schnittstellen für diese Verbindung:

- RS-232 Schnittstelle
- Ethernet-Schnittstelle
- JTAG oder JTAG-ähnliche Schnittstellen wie z.B. Freescale BDM
- USB-Schnittstelle
- CAN-Schnittstelle
- Emulator-Schnittstelle

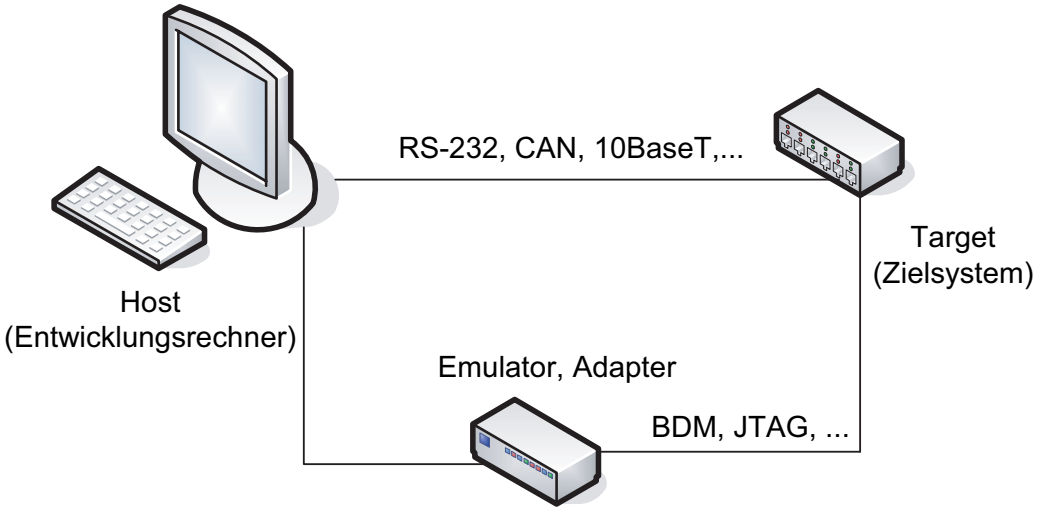


Abbildung 2-1: Entwicklungs- und Zielsystem

Manche dedizierte Systeme besitzen keine dieser Schnittstellen. In einem solchen Fall Verwendung einer ***zusätzlichen Leiterkarte,*** die bei Serienreife wieder entfernt wird.

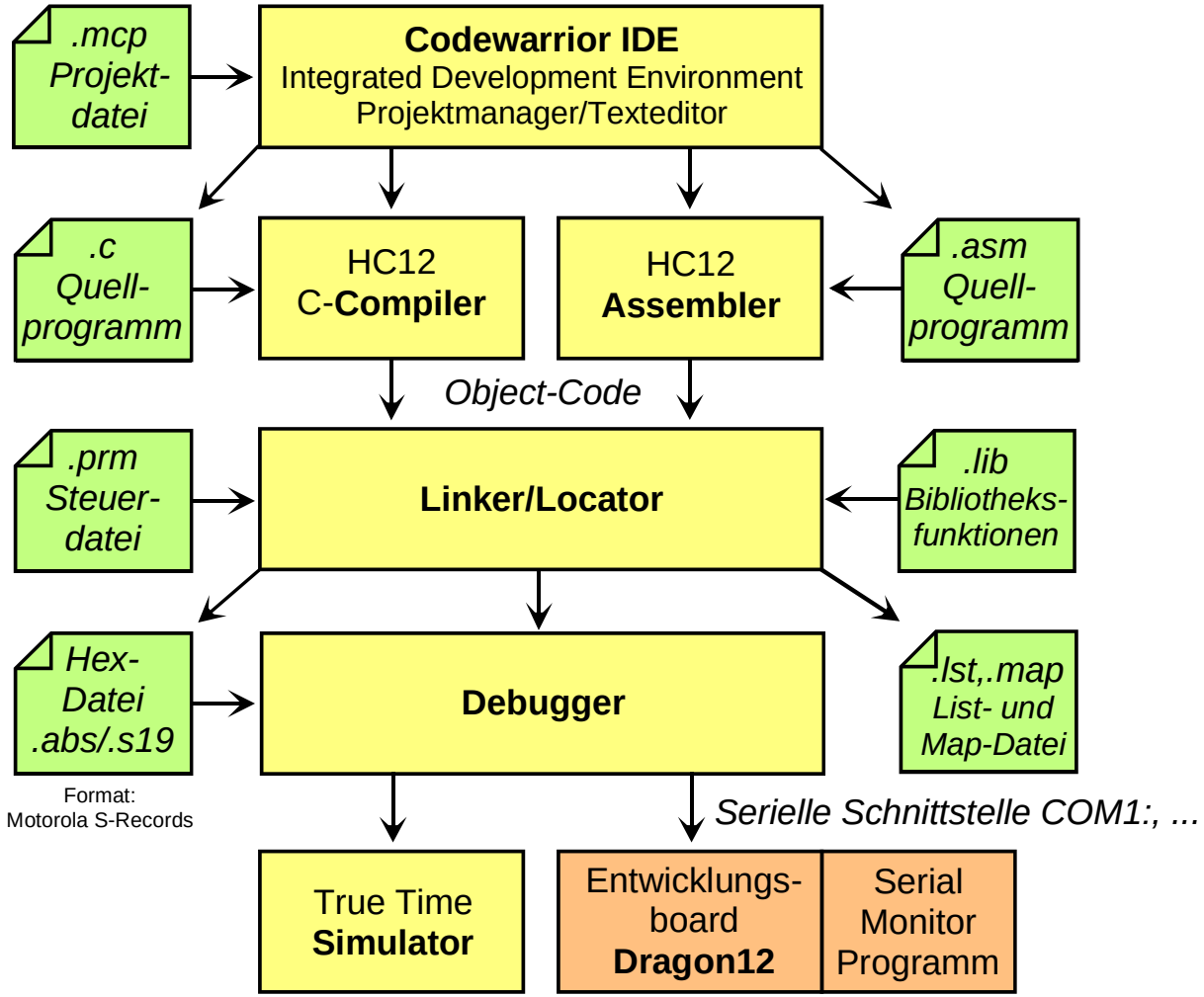
Standard-Entwicklungsrechner stellen heute USB-Anschlüsse und Ethernet-Anschlüsse sowie mit Einschränkungen RS-232-Schnittstellen zur Verfügung. Um JTAG-, BDM-, CAN- oder Emulator-Schnittstellen zu benutzen, ist ***zusätzliche Adapter-Hardware*** erforderlich.

Die Programmspeicher der meisten dedizierten Systeme werden durch System selbst programmiert (***In Circuit Programmierung***).

Das Programmieren mit ***Programmiergeräten*** ist heute die Ausnahme. Bei sehr kleinen kostensensitiven System ***Maskenprogrammierung***.

2.2.3 Vom Quellcode zum ausführbaren Maschinencode

Aus Grundlagenvorlesung bekannt: editieren, kompilieren, linken, ausführen:



Besonderheit bei dedizierten Systemen: der *Locating-Prozess* und *Linker-Befehlsdatei*.

2.2.4 Der Link- und Locating-Prozess

Bei *Desktop-Systemen*:

1. Am Ende des Entwicklungsprozesses ausführbare Datei (*Image-Datei*, oder *Image*), z.B. `a.out` oder `meinProgramm.exe`.
2. Starten eines Programms durch Doppelklicken oder Aufruf der Image-Datei von Kommandozeile aus.
3. *Betriebssystem* lädt die ausführbare Datei irgendwo in den Hauptspeicher
4. Dabei werden eventuell vorher noch nicht bekannte Adressen angepasst.
5. Einsprungspunkt des Programms wird angesprungen.

Desktop- und Serversysteme: vor Startvorgang noch nicht bekannt, wohin in Hauptspeicher Programm geladen wird. Benutzer und Programierer müssen sich keine Gedanken darüber machen, wo das Programm laufen wird.

Bei *dedizierten Systemen*:

- Meist vorher bekannt, wo im Speicher Anwendung laufen soll
- Bekannt, wieviel und welche Art Speicher wo zur Verfügung steht (*memory map*)

Zwei Strategien zum Abbilden des Maschinencodes auf das Zielsystem:

- Adressen werden *vor dem Laden* ins Zielsystem *festgelegt*. Erlaubt Format mit absoluten Adressen für das ausführbare Programm (z.B. Intel-Hex oder Motorola-S-Record).
- das Zielsystem hat einen Lader, der die *Adressen beim Starten* des *Systems* bestimmt. Erfordert Format mit *relokierbaren Adressen* für das ausführbare Programm (z.B. ELF, COFF, PE).

Erste Methode vor allem bei kleineren Systemen, da geringste Anforderungen ans Zielsystem.

Zweite Methode erfordert *Lader*, der Adressen beim Laden einsetzen kann.

2.2.5 Die Linker-Befehlsdatei

Dedizierte Systeme teilen Adressraum auf verschiedene Arten von Speichertypen auf:

- *Festwertspeicher* (meist Flash-Speicher, oder EPROM), enthält Programmcode, der beim Einschalten des Systems ablaufen muss. Da dedizierte Systeme meist keine Festplatten besitzen, müssen Anwendungsprogramme ebenfalls in Festwertspeicher (meist Flash-Speicher) abgelegt sein.
- *Schreib-Lesespeicher* mit wahlfreiem Zugriff (RAM)
- *Nichtflüchtigen Speicher* (meist EEPROM) für Konfigurationsdaten

Aufteilung der Adressbereiche ist spezifisch für jeden Typ eines dedizierten Systems.

Compiler- bzw. Linkerhersteller kann nicht wissen, wie Verteilung im konkreten Fall aussieht (im Gegensatz zum PC)

Programmierer muss Linker anweisen, die Adressen für die verschiedenen Daten und Programmteile korrekt zu bestimmen. Diese Anweisungen werden in einer *Linker-Befehlsdatei* zusammengefasst.

Linker-Befehlsdateien sind *nicht standardisiert*: Befehle und Formate abhängig von Entwicklungsumgebung.

In der Praxis aber große Ähnlichkeit.

Abschnitte (Sections)

Während des Compiliervorgangs: Compiler ordnet Code und Daten bestimmten *Abschnitten* (*sections*) zu.

Diese Zuordnung lässt sich z.T. mit `pragma`-Direktiven steuern.

Assemblerprogrammierer kann Zuordnung bei komfortableren Systemen selbst vornehmen.

Die *Anzahl* und *Art* von Abschnitten ist *nicht standardisiert*, es gibt aber Standards wie ELF (executable and linking format), die für sich Vorgaben machen.

Tabelle 2.1: Beispiel für ELF-Standard mit *Abschnittstypen*:

- NOBITS heißt, dass in der Image-Datei keine Daten für diesen Abschnitt enthalten sind.
- PROGBITS bezeichnet Abschnitte, für die in der Image-Datei Daten oder Maschinenbefehle enthalten sind. Anwendungsprogrammier können auch eigene Abschnitte definieren.

- DYNAMIC für dynamisches Linken (shared libraries, DLL)

Listing 2-1 zeigt eine ***Linker-Befehlsdatei*** für ein dediziertes System mit 16-Bit-Prozessor von Freescale, den wir in dieser Vorlesung verwenden.

Tabelle 2.1: Im ELF-Standard definierte spezielle Abschnittstypen

Abschnitt	Typ	Beschreibung
.bss	NOBITS	nicht initialisierte Daten
.comment	PROGBITS	z.B. zur Versionkontrolle
.data und .data1	PROGBITS	initialisierte Daten
.debug	PROGBITS	Informationen für symbolisches Debuggen
.dynamic	DYNAMIC	Symboltabelle für dynamisches Linken
.hash	HASH	Symbol-Hashtabelle
.line	PROGBITS	Zeilennummer-Information für symbolisches Debuggen
.note	NOTE	Anmerkungen zum File
.rodata und .rodata1	PROGBITS	Nur-Lesedaten
.shstrtab	STRTAB	Abschnittsnamen
.strtab	STRTAB	Namen der Symboltabelle
.symtab	SYMTAB	Symboltabelle
.text	PROGBITS	Ausführbarer Code

Auch hier gibt es vordefinierte Sections (z.B. `ROM_VAR` für Konstanten oder `STRINGS` für Zeichenketten in den Zeilen 12 und 13).

```
1  NAMES END
2
3  SEGMENTS
4      RAM = READ_WRITE 0x1000 TO 0x3FFF;
5      ROM_4000 = READ_ONLY  0x4000 TO 0x7FFF;    /* unbanked FLASH ROM */
6      ROM_C000 = READ_ONLY  0xC000 TO 0xFEFF;
7  END
8
9  PLACEMENT
10     _PRESTART,               /* jump to _Startup at the code start */
11     STARTUP,                 /* startup data structures */
12     ROM_VAR,                 /* constant variables */
13     STRINGS,                 /* string literals */
14     VIRTUAL_TABLE_SEGMENT,   /* C++ virtual table segment */
15     DEFAULT_ROM, NON_BANKED, /* runtime routines which must not be banked*/
16     COPY                     INTO  ROM_C000;
17     OTHER_ROM                INTO  ROM_4000;
18     DEFAULT_RAM              INTO  RAM;
19 END
20
21 STACKSIZE 0x100
22
23 VECTOR 0 _Startup           /* reset vector: default entry point
24                                for C/C++ application. */
```

Listing 2-1: Linker-Befehlsdatei der Codewarrior-Entwicklungsumgebung

Während des *Locating-Vorgangs* werden die *Sections* sogenannten *Segmenten zugeordnet*.
Segmente bezeichnen Adressbereiche im Speicher.
In der Linkerbefehlsdatei in Listing 2-1 drei Segmente (Zeilen 4 bis 6):

- RAM für Schreiblese-Speicher im Adressbereich von 0x1000 bis 0x3FFF
- Festwertspeicher von 0x4000 bis 0x7FFF
- Festwertspeicher 0xC000 bis 0xFEFF

Diese Definition leitet sich von der „*Memory Map*“ ab, deren Kenntnis bei der Entwicklung dedizierter Systeme wichtig ist.

Memory Map

Jedes dedizierte System besitzt spezifische Verteilung verschiedener *Speicherarten* (Festwertspeicher, Schreib-Lesespeicher, EEPROM, Peripherieregister)

Art des Speichers muss bei der Systeminitialisierung, beim Compiliervorgang und beim Linkvorgang berücksichtigt werden

Bei Initialisierung des Systems *darauf achten*, dass *Schnittstellenregister* wie z.B. für parallele Ein- und Ausgabe *von* einem eventuell vorhandenen *Cashing-Management ausgenommen* werden. Werden solche Bereiche zur schnelleren Ausführung in einen Cash-Speicher geladen, wird die Verbindung zur Außenwelt unkontrollierbar für bestimmte Zeitabschnitte unterbrochen.

Beim Compiliervorgang *darauf achten*, *dass Compiler Schreib- und Lesevorgänge auf Schnittstellenregister nicht optimiert*, z.B. indem er ein Schnittstellenregister einmal in ein prozessorinternes Register lädt und den Wert des Registers danach nur noch von dort holt. Einen entsprechenden *Hinweis* gibt man dem Compiler in C *mit* dem Schlüsselwort *volatile.*

Die Memory Map für den in dieser Vorlesung verwendeten 68HCS12-Freescale-Rechner. Siehe auch Vorle-

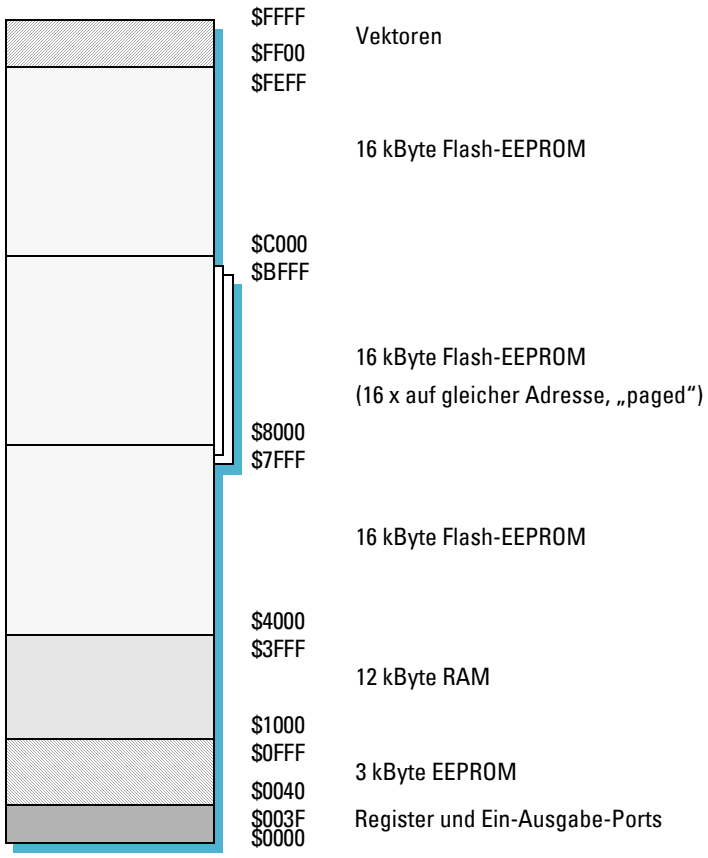
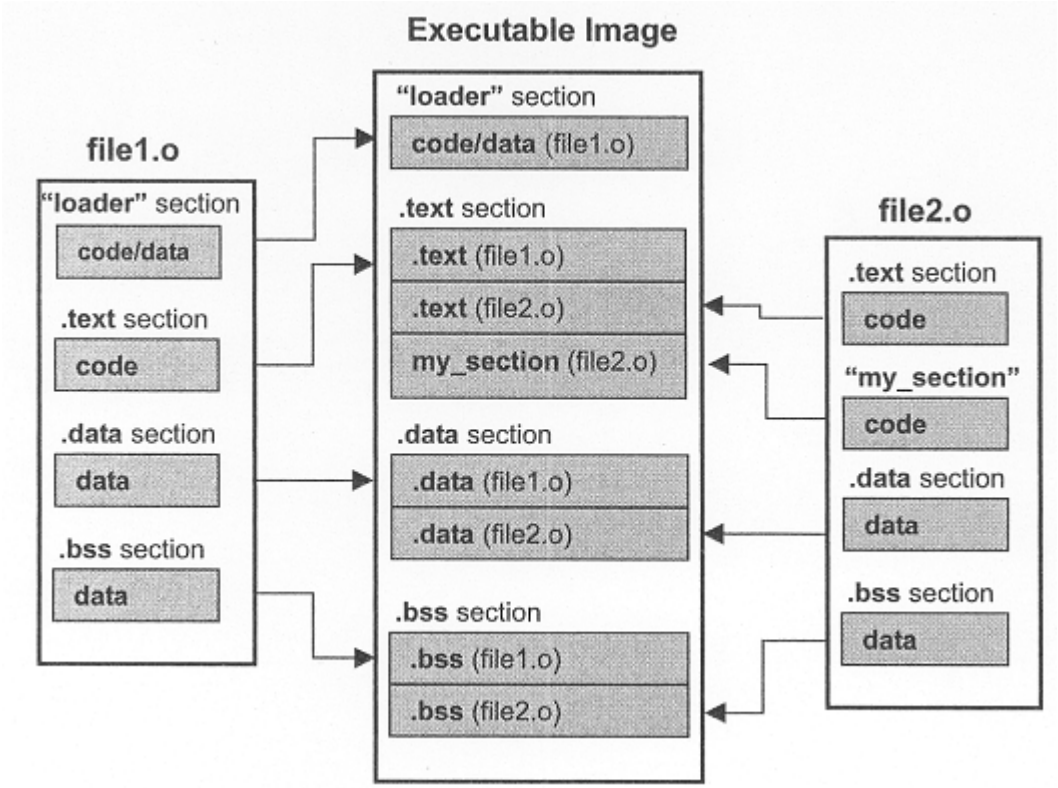


Abbildung 2-2: Memory-Map für den Freescale-Microcontroller 68HCS12

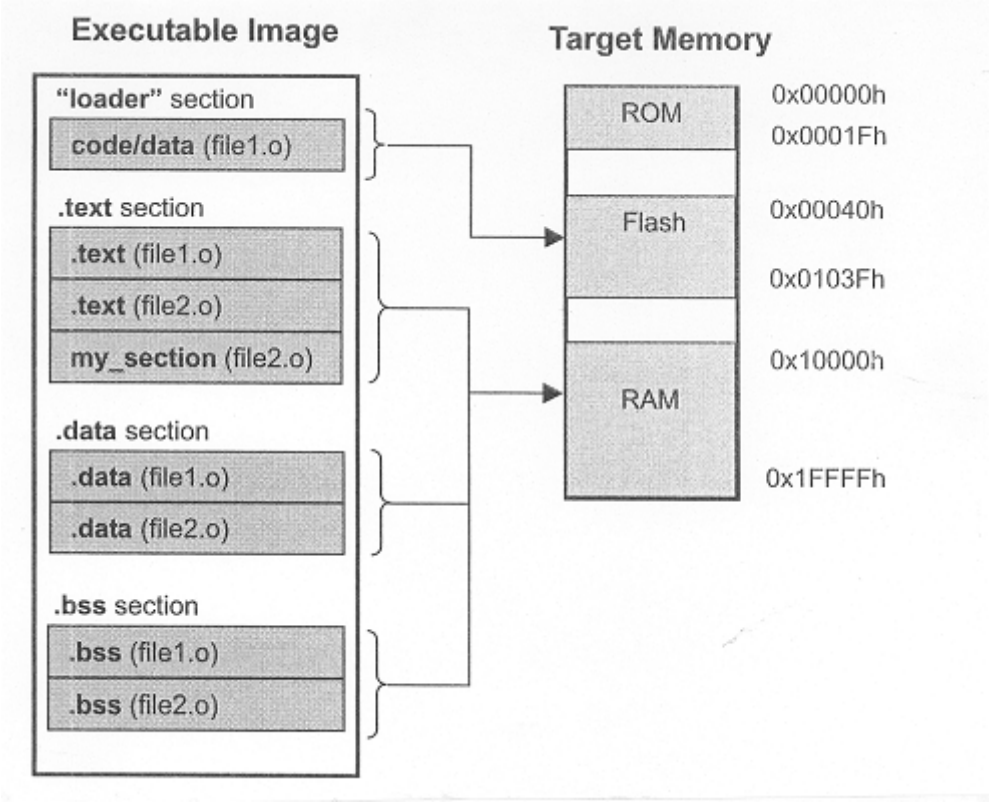
sung „Computerarchitektur 3“.

Beim Linkvorgang muss der Linker die gleichartigen Abschnitte der beim Kompilieren entstandenen einzelnen Objektdateien zusammenfassen.



Beispiel: zwei *Objektdateien* werden zu einem *Executable* zusammengeführt.

Für den Fall, dass eine *Image-Datei* mit *absoluten Adressen* erzeugt werden soll, muss ein *Locator* die einzelnen Abschnitte Speicherbereichen (Segmenten) zuordnen. Dieser Vorgang wird durch die Befehle in der Linkerbefehlsdatei gesteuert (Zeilen 1-9 oben)



2.2.6 Image-Dateiformate mit absoluten Adressen

Es gibt zwei dominante Dateiformate für Image-Dateien mit absoluten Adressen:

- *S-Record-Format* der Firma Motorola
- *Hex-Format* der Firma Intel

Beide Formate kodieren binäre Information als ASCII-Zeichen.

Motorola S-Record-Format (S19)

Eine S-Record-Datei besteht aus einer Reihe speziell formatierter Zeilen (Records) verschiedener Länge. Die Reihenfolge der Zeilen ist nicht definiert; sie enthalten Zahlen in hexadezimaler ASCII-Darstellung. Die Zeilen sind nach dem in 2-3 gezeigten Schema aufgebaut.

Abbildung 2-3: Aufbau eines S-Records

Der Typ eines Records bestimmt, wie die restlichen Zeichen der Zeile zu interpretieren sind. Wichig zum Verständnis sind die folgenden Record-Typen

- 'S1': das Adressfeld wird als 2-Byte-Adresse interpretiert. Die Daten sind an diese Adresse in den Speicher zu laden.
- 'S2' und 'S3': Wie 'S1', außer dass das Adressfeld als 3-Byte bzw. 4-Byte-Adresse interpretiert wird.
- 'S9': das Adressfeld beinhaltet die 2-Byte lange Startadresse des Programms.
- 'S7' und 'S8': wie 'S9', nur für 4-Byte bzw. 3-Byte lange Adressen.

Das folgende Listing zeigt ein kleines Assemblerprogramm für einen Freescale 68HCS12-Rechner.

```
1  Loc    Obj. code   Source line
2  ------ ---------   -----------
3  000000 CFxx xx             lds  #stack+$100
4  000003 86FF                ldaa #$ff
5  000005 5A03                staa DDRB          ; Data Direction Register B
6  000007 180B FF02           movb #255, DDRP    ; Data Direction Register P
7  00000B 5A
8  00000C 7A02 58             staa PTP           ; alle LEDs ausschalten
9
10                    loop:
11 00000F 180B 01xx           movb #1,platzhalter ; gib „H“ auf LED aus
12 000013 xx
13 000014 180B 7600           movb #$76,PORTB     ;
14 000018 01
15 000019 180B 0E02           movb #%1110, PTP    ;
16 00001D 58
17 00001E 20EF                BRA loop           ; Endlosschleife
```

Das nächste Listing zeigt die ladbare *S-Record-Datei*, die aus diesem Programm entstanden ist.

```
1  S123C00010EFCF110186FF5A03180BFF025A7A0258180B011000180B760001180B0E0258AF
2  S105C02020EF0B
3  S105FFFEC0003D
4  S9030000FC
```

Zeile 1: Programm an die Adresse 0xC000 und aufwärts geladen.
Die ersten Bytes an dieser Adresse entsprechen den Bytes in den Zeilen 3 bis 5 des Assemblerprogramms.
Zeile 3: Die Adresse 0xFFFE, die hier mit dem Wert 0xC000 geladen wird, beinhaltet den *Reset-Vektor*. Dort wird beim Einschalten des Rechners mit der Programmausführung begonnen.
Zeile 4: S9-Record hat hier keine Bedeutung, die Startadresse wird in Zeile 3 festgelegt.

Intel HEX-Format

Wie das Motorola S-Record-Format ist das Intel HEX-Format ein *textbasiertes Format*, um Daten auf Speicherbereiche abzubilden. Jede Zeile besteht aus hexadezimalen Werten, die eine Adresse bzw. einen Adressoffset und die dazu gehörenden Daten beschreiben.

Neben der ursprünglichen Form des Intel HEX-Formats für 8-Bit-Rechner gibt es zwei Erweiterungen, die vor allem das Problem der erweiterten Adressierung bei den 16-Bit und 32-Bit-Prozessorarchitekturen beheben.

Die folgende Abbildung zeigt den Aufbau einer Zeile im Intel HEX-Format.

Im Intel HEX-Format 8-Bit gibt es nur zwei Typen von Zeilen:

- 00: *Daten-Record*, enthält Daten und 16-Bit-Adressen.
- 01, *End Of File Record*, beendet eine Datei. Muss die letzte Zeile einer Datei sein und enthält keine Daten. Sieht in der Regel so aus: `':00000001FF'`.

Die 16-Bit und 32-Bit-Varianten fügen den beiden oben beschriebenen Zeilentypen vier weitere hinzu. Die Zeilentypen 02 und 03 unterstützen 16-Bit segmentierte Adressierung, die Zeilentypen 04 und 05 unterstützen 32-Bit lineare Adressierung.

2.2.7 Image-Dateiformate mit relokierbaren Adressen

Damit ein Programm auf einem dedizierten System ablaufen kann, müssen alle im Programm verwendeten Adressen wie z.B. Aufrufe von Unterfunktionen mit den tatsächlichen Gegebenheiten im Zielsystem übereinstimmen, sie müssen absolut festgelegt sein.

Der *Linker erzeugt* eine *Image-Datei*, die noch nicht festgelegte Adressen enthalten kann. Für kleinere dedizierte System verwendet man auf dem Entwicklungsrechner einen *Locator*, um diese Adressen vor dem Laden des Programms auf das Zielsystem festzulegen. Bei größeren dedizierten Systemen ist der Locator auf dem dedizierten System selbst untergebracht, z.B. als Teil eines dort schon laufenden Betriebssystems.

Der Locator muss das Format der vom Linker erzeugten Image-Datei kennen, um das zugehörige Programm den richtigen Speicheradressen zuordnen zu können. Neben einer Anzahl proprietärer Formate für die vom Linker erzeugte Datei gibt es drei weit verbreitete Formate, die hier etwas näher beschrieben werden sollen.

Executable und Linking Format (ELF)

ELF ist ein weit verbreitetes Objektdatei-Format auf Unix-Betriebssystemen und Entwicklungsumgebungen für dedizierte Systeme. So nutzt zum Beispiel die Entwicklungsumgebung für die Sony Playstation ELF. Die in den Übungen verwendete Codewarrior-Entwicklungsumgebung kann konfiguriert werden, ELF-Objektdateien zu erzeugen und zu verarbeiten. Die RMOS-Entwicklungsumgebung arbeitet ebenfalls mit ELF.

ELF unterstützt drei Arten von Objektdateien:

- Eine *relokierbare Datei* enthält Code und Daten und ist dazu gedacht, mit anderen Objektdateien zusammengebunden zu werden, um eine ausführbare Datei oder eine gemeinsam nutzbare (shared) Objektdatei zu bilden.
- Eine *ausführbare Datei (executable)* enthält ein ablauffähiges Programm.
- Eine *gemeinsam nutzbare Objektdatei (shared object file)* enthält Code und Daten, die entweder von einem Linker mit anderen Objektdateien zu einer neuen Objetkdatei zusammengebunden werden kann, oder die von einem dynamischen Linker mit einer ausführbaren Datei und anderen gemeinsam nutzbaren Objetkdateien zu einem ablauffähigen Image zusammengeführt werden kann.

ELF-Dateien unterstützen sowohl den Vorgang des Bindens als auch den des Ladens zu einem ausführbaren Prozess-Image. Sie bestehen aus

- einem Header,
- null oder mehr Segmenten

- null oder mehr Abschnitten (sections)

Die Segmente enthalten Daten, die für das Laden und anschließende Ausführen des Codes hilfreich sind. Abschnitte (sections) enthalten Information, die das *Binden* und die *Relokation* unterstützen. Die in einer Objektdatei vorhandenen Segmente und Sections werden in einer *Programm-Header-Tabelle* bzw. einer *Section-Header-Tabelle* beschrieben.

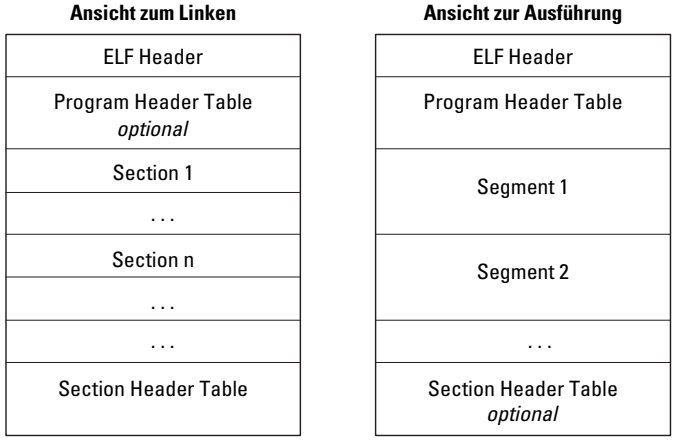


Bild 2.4: ELF-Dateistruktur

Common Object File Format (COFF)

Common Object File Format (COFF) ist ein Format für ausführbare Dateien, Objektdateien und Shared Libraries.

Kommt von Unix System V, ist später von Microsoft mit leichten Veränderungen übernommen worden.

Ist in vielen Systemen inzwischen *durch* das *ELF ersetzt* worden.

War selbst Ersatz für das sehr (zu) einfache a.out Format (das keine Untersrützung für shared libraries und symbolisches Debuggen bot).

COFF führte Sections ein, aber:

- Zahl der Sections war begrenzt
- Section-Namen waren zu kurz
- Debug-Unterstützung nur für C

Portable Executable (PE) Format

Portable Executable-Format wird für ausführbare Dateien, Objektdateien und DLLs in Windows-Betriebssystemen einschließlich *Windows CE* verwendet.

Das PE-Format basiert auf dem COFF-Dateiformat aus der Unix-Welt. Es unterstützt die x86-Rechnerarchitektur, ARM9, Renesas SH4 sowie MIPS.

PE-Dateien enthalten keinen *Code ohne Adresszuweisungen*. Das Programm wird für eine *bevorzugte Basisadresse* kompiliert und gelinkt. Kann ein PE-basiertes Image nicht an die bevorzugte Adresse geladen werden, muss der Lader alle Adressen verschieben.

- Wenn keine Neuberechnung der Adressen erforderlich ist, kann Image so sehr schnell zu Ausführung kommen;
- wenn aber eine Verschiebung der Adressen notwendig ist, dauert der Ladevorgang recht lang.

Problematisch wird es auch, wenn DLLs nicht an die bevorzugte Adresse geladen werden können. In diesem Fall müssen sie mehrfach geladen werden, was ihrem Zweck widerspricht. Im ELF-Verfahren ist der gesamte Code immer beim Laden mit Adressen zu versehen.

Dadurch dauert Ladevorgang länger als bei PE, aber Speicher wird u.U. effektiver genutzt.

2.2.8 Zielsystem-Monitorprogramm

Während der Entwicklungsphase: Programmcode muss für Test schnell vom Entwicklungssystem auf das Zielsystem geladen werden können.

Dazu müssen auf Zielsystem vorhanden sein:

- eine entsprechende Hardwareschnittstelle
- die zur Bedienung der Schnittstelle notwendige Software

Es ist auch vorstellbar, dass diese Software weitere Funktionalität zur Verfügung stellt, um das Zielsystem zu steuern und zu beobachten.

Ein entsprechendes Programm nennen wir „*Monitor*“-Programm. Wir können Monitor-Programme in drei Klassen aufteilen:

- einfache *Lader*
- *Monitorprogramm mit Benutzerschnittstelle*
- *Zielsystem-Debugmonitore*

Monitorprogramme mit Benutzerschnittstelle und Zielsystem-Debugmonitore beinhalten in der Regel ein Lader-Programm, mit dem man lediglich Programmcode auf das Zielsystem laden kann. *Monitore* werden verwendet, *wenn noch kein Betriebssystem* auf der Zielplattform zur Verfügung steht, dass eine vergleichbare Unterstützung bietet.

Monitore sind selbst auch Programme, müssen entwickelt werden und auf das Zielsystem geladen werden. Man benötigt für das *erste Laden* eines Monitors auf das Zielsystem *spezielle Adapter*.

Bei der Entwicklung von Monitoren leisten *Emulatoren* (In Circuit Emulatoren, *ICEs*) eine gute Hilfe, da sie einen Einblick in das System ohne Softwareunterstützung durch das System selbst erlauben. Emulatoren sind auch hilfreich bei der Erstinbetriebnahme von Rechner-Hardware.

Monitorprogramme enthalten häufig den *Initialisierungscode* (*boot code*) für das dedizierte System, und verbleiben auch in den serienreifen Geräten. Debugmonitore müssen speziell auf den verwendeten Debugger auf dem Entwicklungssystem abgestimmt sein. Sie nützen die Möglichkeiten des Mikrorechners, *Breakpoints* zu setzen und zu löschen sowie weitere eventuell zur Verfügung stehende Debug-Leistungsmerkmale.

Monitorprogramme bedienen die Schnittstelle zum Entwicklungssystem. Sie können sehr einfach gehalten werden und z.B. nur eine RS-232-Verbindung unterstützen. Sie können aber auch sehr komplex werden und

z.B. eine TCP/IP-Verbindung bedienen; in diesem Fall muss schon beim ersten Laden eines Programms in das Zielsystem auf diesem ein kompletter Kommunikationsstack funktionsfähig sein.

2.3 Systeminitialisierung

Beim Starten eines Rechners beginnt die CPU an einer bestimmten Adresse mit der Ausführung von Maschinenbefehlen.

Diese Adresse nennt man den *Reset-Vektor*. Er oder die Stelle, wo er abgespeichert ist, sind durch die Hardware vorgegeben.

An dieser Adresse muss beim Einschalten des Systems Maschinencode stehen (Festwertspeicher).

- Bei PC: BIOS des Hauptplatinenherstellers
- Bei dediziertem System: eigener Code.

Schritt 1:

- Dieser Code muss Hardware initialisieren, so dass Software laufen kann (z.B. Bus-Konfiguration, Waitzyklen, Peripherie auf Ein- oder Ausgang schalten etc.): *Boot Code*.

Schritt 2:

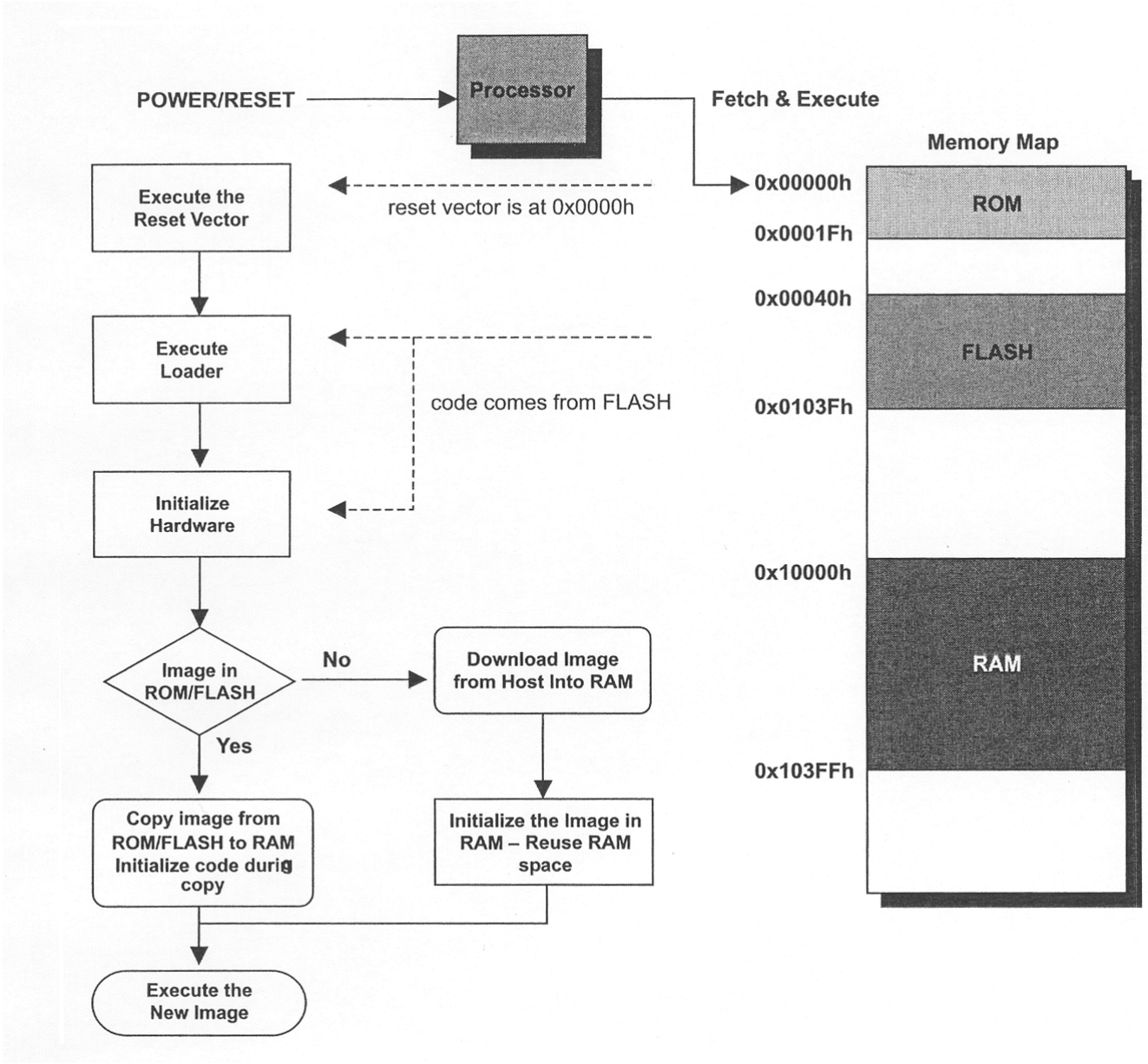
- Wenn Betriebssystemsystem vorhanden: Betriebssystem laden.

Schritt 3:

- Anwendung laden, wenn sie nicht schon an richtiger Stelle im Adressraum liegt.

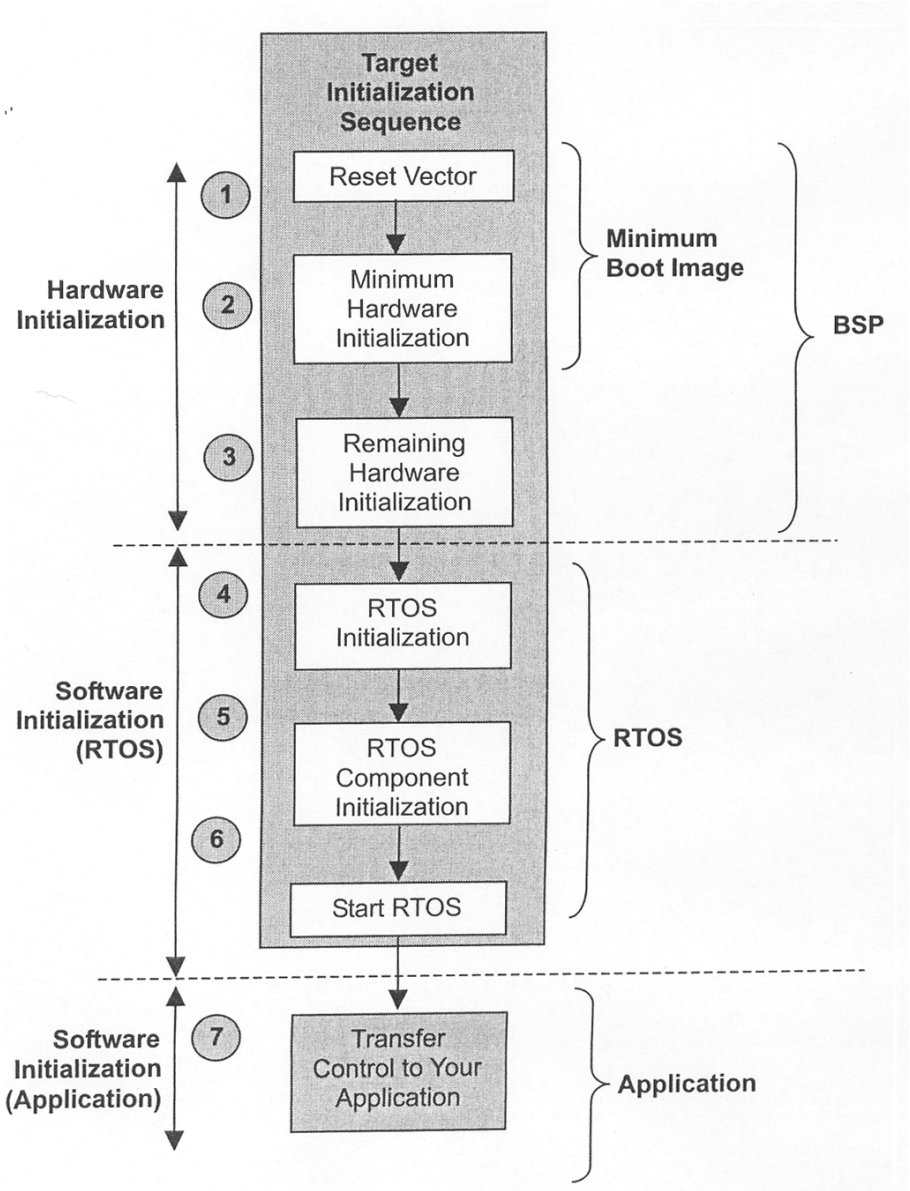
Schritt 4:

- Anwendung starten.


POWER/RESET
Processor
Fetch & Execute
Memory Map
Execute the Reset Vector
reset vector is at 0x0000h
0x00000h
ROM
0x0001Fh
0x00040h
Execute Loader
FLASH
code comes from FLASH
0x0103Fh
Initialize Hardware
0x10000h
Image in ROM/FLASH
No
Download Image from Host Into RAM
RAM
Yes
0x103FFh
Copy image from ROM/FLASH to RAM Initialize code durig copy
Initialize the Image in RAM – Reuse RAM space
Execute the New Image

Anmerkungen:

- Die *Anwendung liegt im Festwertspeicher*, und kann zum Ausführen aus Geschwindigkeitsgründen ins RAM geladen werden. Der dazu notwendige Lader ist Teil des Boot Codes oder des Betriebssystems. *Nur bei größeren Systemen mit viel RAM.*
- Der Boot-Code selbst kann aufgeteilt sein in einen nicht änderbaren Teil und einen durch Softwaredownload austauschbaren Teil.
- In manchen Systemen kann der gesamte Boot-Code ausgetauscht werden. Mögliches Problem: Fehlfunktion (z.B. Power Fail) während Austauschvorgang.


Target Initialization Sequence
Reset Vector
Minimum Hardware Initialization
Remaining Hardware Initialization
RTOS Initialization
RTOS Component Initialization
Start RTOS
Transfer Control to Your Application
1
2
3
4
5
6
7
Hardware Initialization
Software Initialization (RTOS)
Software Initialization (Application)
Minimum Boot Image
BSP
RTOS
Application

2.4 Modellgetriebene Softwareentwicklung

Höhere Produktivität bei der Softwareentwicklung hauptsächlich durch

- Wiederverwendung bestehender Lösungen
- Erhöhung des Abstraktionsgrades bei der Beschreibung der Softwarelösung

Mit dem Ansatz der modellgetriebenen Softwareentwicklung nutzt man beide Möglichkeiten.
Aufteilung der Modellebenen in

- Fachliche (plattformunabhängige) Modelle
- Technische (plattformabhängige) Modelle

Mit einem *Transformator* wird ein fachliches Modell in ein technisches überführt.
Die Entwicklung geht immer vom Modell aus. Änderungen fließen zuerst in das Modell ein, bevor die Prozesskette weitergeführt wird.
Einige Werkzeuge im Echtzeitbereich:

- Matlab/Simulink
- ASCET von ETAS

2.5 Hardware in the Loop (aus Wikipedia)

Hardware in the Loop (*HiL*) bezeichnet ein Verfahren, bei dem ein eingebettetes System (z. B. reales elektronisches Steuergerät oder reale mechatronische Komponente) über seine Ein- und Ausgänge an ein angepasstes Gegenstück angeschlossen wird und dadurch den Kreis (Loop) schließt. Dieses Gegenstück wird i. A. HiL-Simulator genannt und dient als Nachbildung der realen Umgebung des Systems. Hardware in the Loop ist eine Methode zum Testen und Absichern von eingebetteten Systemen, zur Unterstützung während der Entwicklung sowie zur vorzeitigen Inbetriebnahme von Maschinen und Anlagen.
Dabei wird das zu steuernde System (z. B. Auto) über Modelle simuliert, um die korrekte Funktion des zu entwickelnden Steuergerätes (z. B. Motorsteuergerät) zu testen.

Die Eingänge des Steuergeräts werden mit Sensordaten aus dem Modell stimuliert. Um die Reglerschleife (Loop) zu schließen, wird die Reaktion der Ausgänge des Steuergeräts, z. B. das Ansteuern eines Elektromotors, in das Modell zurückgelesen.

Die HIL-Simulation muss meist in Echtzeit ablaufen und wird in der Entwicklung benutzt, um Entwicklungszeiten zu verkürzen und Kosten zu sparen. Insbesondere lassen sich wiederkehrende Abläufe simulieren. Dies hat den Vorteil, dass eine neue Entwicklungsversion unter den gleichen Kriterien getestet werden kann, wie die Vorgängerversion. Somit kann detailliert nachgewiesen werden, ob ein Fehler beseitigt wurde oder nicht.

Die Tests an realen Systemen lassen sich dadurch stark verringern und zusätzlich lassen sich Systemgrenzen ermitteln, ohne das Zielsystem (z. B. Auto und Fahrer) zu gefährden.

Die HIL-Simulation ist immer nur eine Vereinfachung der Realität, es kann den Test am realen System deshalb nicht ersetzen. Falls zu große Diskrepanzen zwischen der HIL-Simulation und der Realität auftreten, sind die zugrundeliegenden Modelle in der Simulation zu stark vereinfacht. Dann müssen die Simulations-Modelle weiterentwickelt werden.

2.6 Ein paar Regeln zum Programmieren in C

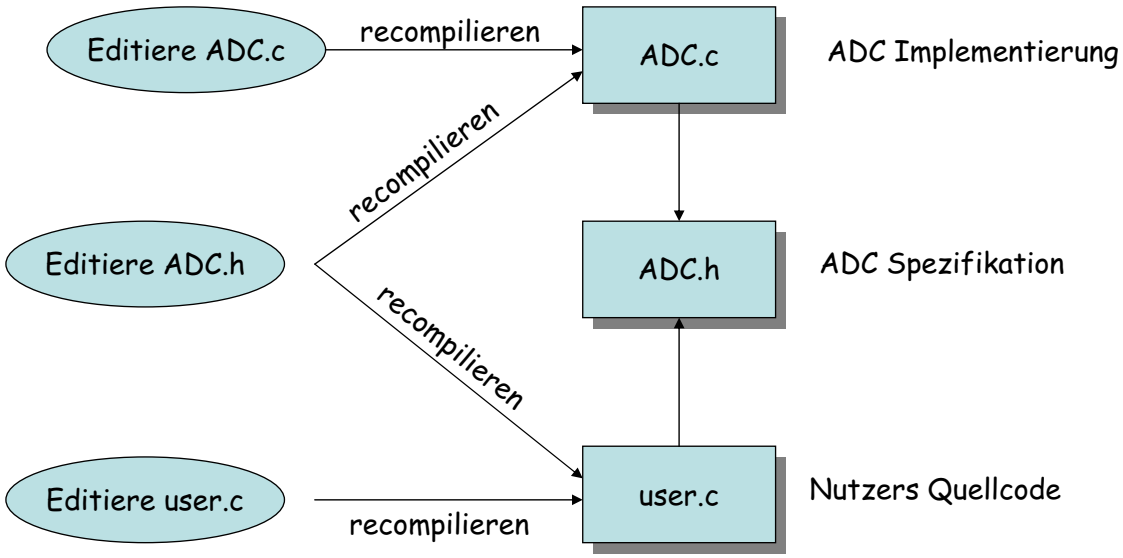
Weitere Regeln der Automotive-Industrie: MISRA (http://www.misra.org.uk). Dokument ist leider nicht frei verfügbar, muss man kaufen. Hier folgt meine persönliche Bestenliste.

2.6.1 Schnittstelle und Implementierung trennen

- Schnittstelle für <modul.c> heisst immer <modul.h>.
- Die beiden Dateien stehen immer im gleichen Verzeichnis, außer bei Bibliotheken
- Alle Funktionen, die nicht in <modul.h> auftauchen, müssen in <modul.c> als „static“ deklariert werden
- Die Beschreibung der Funktionen und Daten der Schnittstelle steht nur in <modul.h>

2.6.2 Abhängigkeiten beim Kompilieren

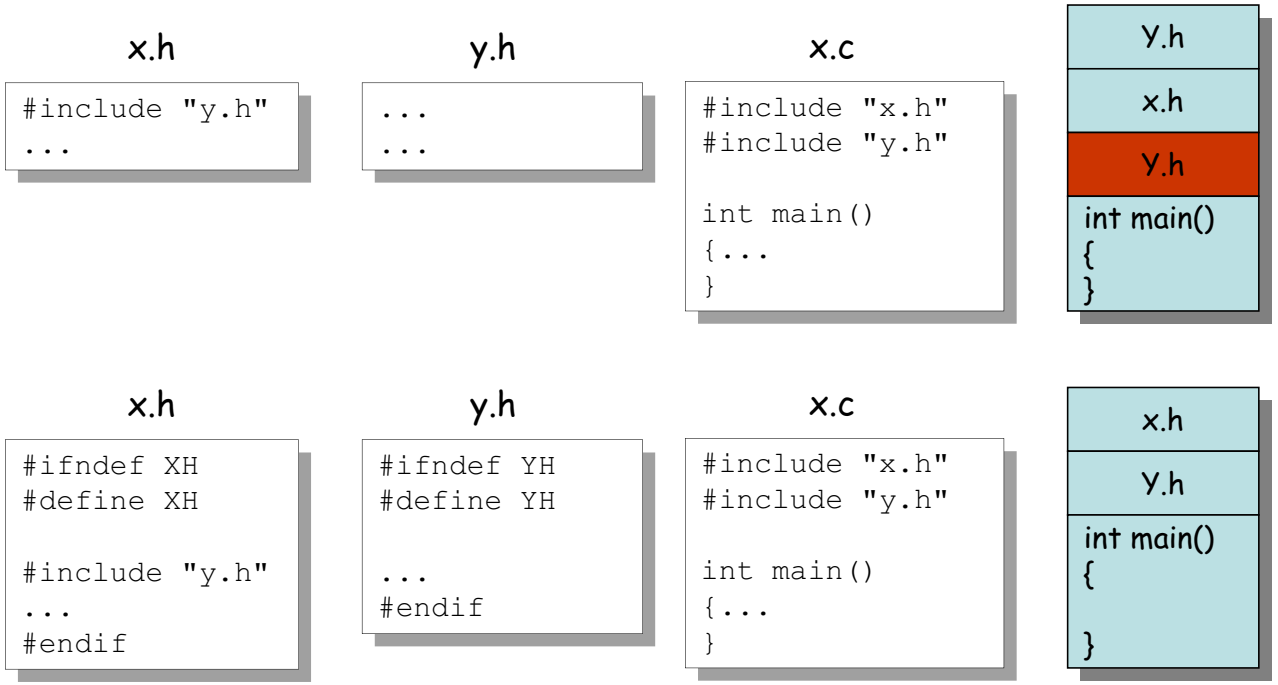
Siehe Darstellung. Wird bei einer IDE meist automatisch gehandhabt.



Abhängigkeiten sind bei manueller Erstellung von make-Dateien zu beachten. Tip: Miller-make ist ein guter Ansatz (http://miller.emu.id.au/pmiller/books/rmch/).

2.6.3 Guards

Header-Dateien *müssen* immer mit *Guards* versehen werden. Guards verhindern, dass in der gleichen Kompiliereinheit Definitionen mehrfach auftreten oder sich unendliche Rekursionen bilden.



2.6.4 Sinnvolle Namen

Geben Sie allen Variablen und Konstanten sinnvolle Namen. Kein normaler Mensch verwendet mehr Namen wie p_ui_count (Zeiger auf unsigned int). Und wie schön liest sich eine Anweisung wie
`if (kellerVoll && kuehlschrankLeer()) {...`

2.6.5 Ein paar weitere Regeln

Hier noch ein paar weitere Regeln, die unbedingt zu beachten sind.

Keine globalen Variablen!

wenn überhaupt, dann z.B. in Modul global.c / global.h und mit Getter- und Setter-Methoden arbeiten

Kein „extern"

„extern" ist nur Valium für den Compiler. Es sagt ihm, dass dieses Symbol irgendwo anders schon irgendwie definiert sein wird, möglicherweise. Einzige Ausnahme: global.c/global.h

Immer alle Warnungen des Compilers und Linkers einschalten!

Alle modul-lokalen Daten und Funktionen mit „static" markieren

2.6.6 Reentrante Funktionen

- Echtzeitsysteme sind meist „Multitasking"-Systeme, d.h. es können mehrere Programme quasi gleichzeitig ablaufen
  - D.h. Programme können sich jederzeit gegenseitig unterbrechen

```
void meinKleinerVertauscher(int *a, int *b)
{   /*  Diese Funktion ist reentrant. */
    int temp = *b;
    *b = *a;
    *a = temp;
}
```

```
static int temp;
void meinSchlechterVertauscher(int *a, int *b)
{   /*  Diese Funktion ist reentrant. */
    temp = *b;
    *b = *a;
    *a = temp;
}
```

2.6.7 Keine Macros

Verzichten Sie, wenn immer es geht, auf Macros.

```
#define aufrundeTeiler(x, y) (x + y - 1) / y
```

benutzt als

```
a = aufrundeTeiler (b & c, sizeof(int));
```

expandiert zu

```
a = (b & c + sizeof(int) - 1) / sizeof(int);
```

ist leider falsch, weil + Vorrang vor & hat !

```
#define min(X, Y) ((X) < (Y) ? (X) : (Y))
```

benutzt als

```
next = min (x + y, foo (z));
```

expandiert zu

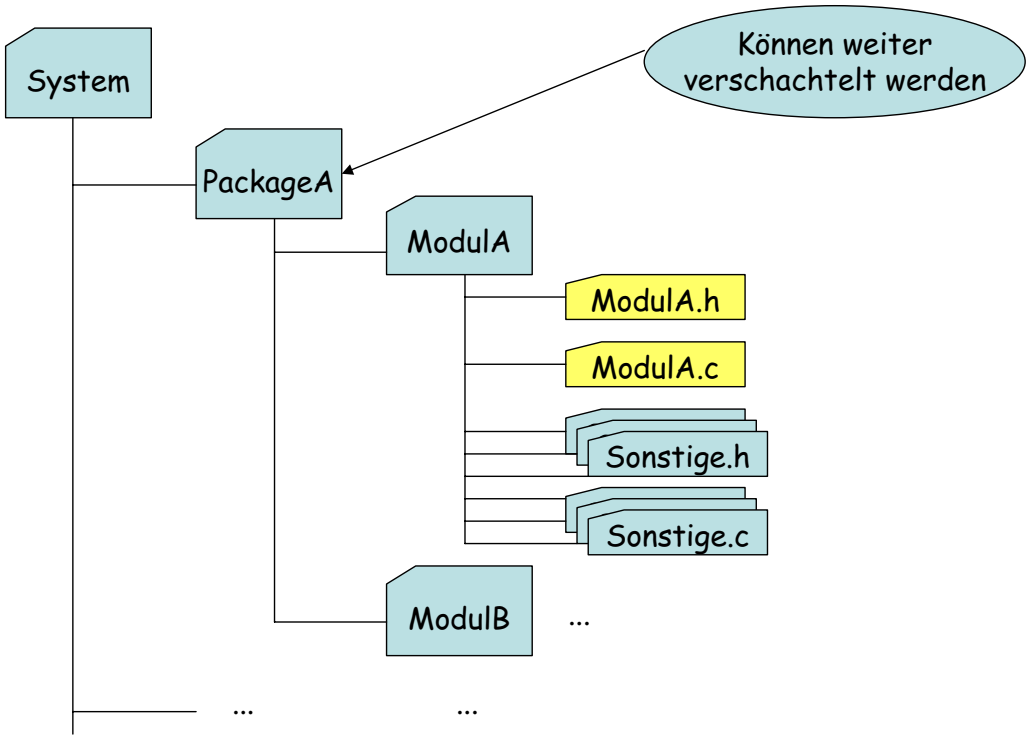
```
next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

bedeutet, dass foo(z) zweimal aufgerufen wird !

Definieren Sie Konstanten nicht mit `#define`, sondern mit `const`. Das erlaubt Typprüfungen, und erleichtert das Debuggen.

2.6.8 Softwarestruktur und Verzeichnisstruktur

Bilden Sie die Softwarearchitektur in die Verzeichnisstruktur ab.



Verwenden Sie keine Pfade in #include-Anweisungen. Die Pfade für die Include-Dateien stehen in der Bauvorschrift bzw. in den IDE-Einstellungen.

3

Hardware und hardwarenahe Programmierung

3.1 Übersicht Schnittstellen

Schnittstellen zur Peripherie verbinden Mikrorechner mit Umwelt. Aufgaben sind

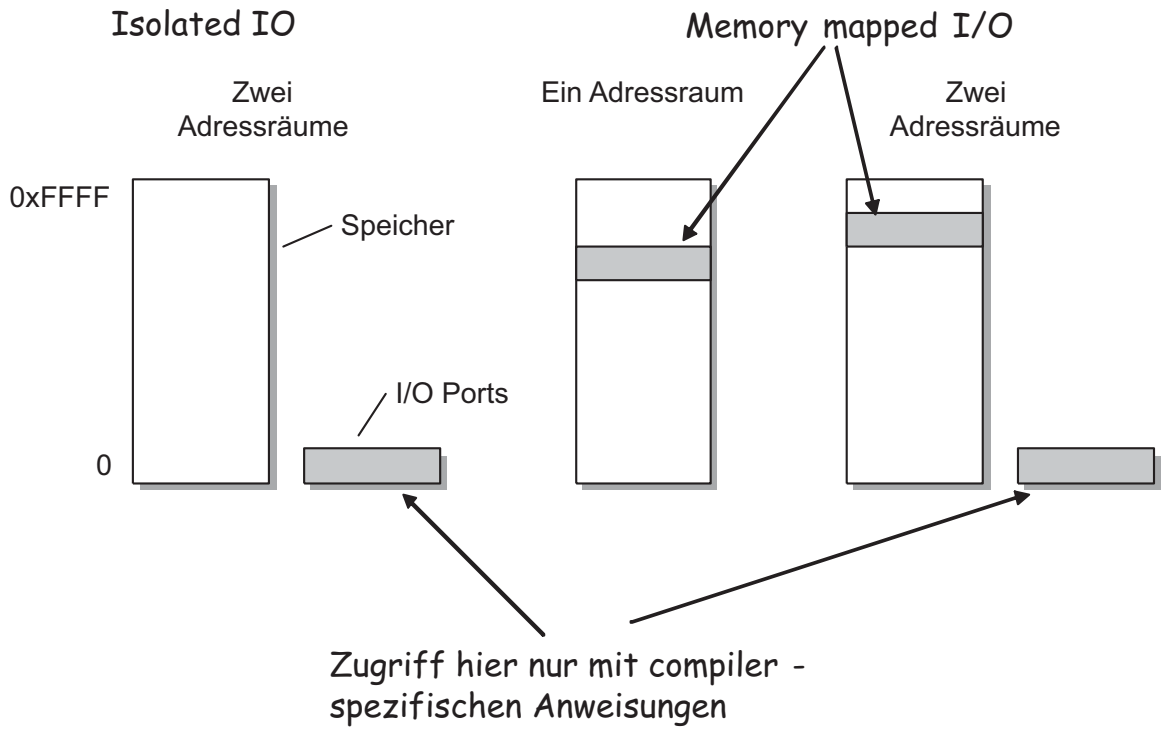
- *Pufferung* von Ein- und Ausgabedaten
- *Umsetzung* von Daten (z.B. seriell/parallel, analog/digital)
- Erzeugung von Steuer- und Handshake-Signalen
- Annahme und Erzeugung von *Unterbrechungsanforderungen*

Schnittstellen sind

- im Mikrocontroller eingebaut
- auf Schnittstellenbausteinen (Interface Controller) realisiert

Integration von Schnittstellen-Funktion und Rechner als

- Speicher-Ein-/Ausgabe (Memory Mapped I/O)
- isolierte Ein-/Ausgabe (Isolated IO)



Beispiel für isolierte IO: Industrie-PC im Labor
Beispiel für Speicher-Ein-/Ausgabe: 68HCS12-Rechnersystem
Klassifizierung von Schnittstellen in

- ***einfache*** Schnittstellen (ein Register für Ein- oder Ausgabe)
- ***komplexe*** Schnittstellen (Eingaberegister, Ausgaberegister, Statusregister, Steueregister, Kontrollregister)
- ***intelligente*** Schnittstellen (mit eigenem Mikrorechner, lokal programmierbar)

3.2 Parallele Ein- und Ausgabe

Praktisch alle Mikrocontroller beinhalten allgemein verwendbare Peripherie-Anschlüsse für Ein- und Ausgabe. Über Steuerregister kann programmiert werden, ob ein Anschluss als Ein- oder Ausgang betrieben werden soll. Meistens sind jeweils acht solcher Anschlüsse zu einer Gruppe zusammengefasst. Diese Gruppe nennt man *Port*.

Abbildung 3-1zeigt die bei unserem 68HCS12-Laborrechner vorhandenen Ports. Der Rechner besitzt acht 8-Bit-Ports (A, B, E, H, M, P, S und T) und zwei kleinere Ports (J und K). Diese Ports können für allgemeine Ein-/Ausgabezwecke verwendet werden, wenn sie nicht für interne Peripheriemodule (Ports H, M, P, S, T und J) bzw. für den Anschluss eines externen Speichers (Port A, B, E und K) verwendet werden.

Die meisten Peripheriemodule haben zusätzliche Fähigkeiten wie z.B. A/D-Wandler, Pulsweitenmodulator und CAN-Schnittstelle. Wir betrachten hier die Eigenschaften der Ports H, M, P, S und T für allgemeine Ein- und Ausgabezwecke.

Die Struktur eines einzigen Pins (von acht) eines solchen Ports ist in Abbildung 3-2 gezeigt.Der Port besitzt Steuerungsregister, RDR*X*, PPS*X* und PER*X* und Datenregister PT*X* und PTI*X*. *X* bezeichnet hier den Port, also H, M, P, S, T oder J. PTI*X* ist nur lesbar und liest immer den Pegel am Pin selbst.

Jedes Bit in diesem Register entspricht einem Anschluss P*Xn* in der Weise, dass Bit *n* im Register zuständig ist für Anschluss P*Xn* des Ports.

Wenn DDR*X*n auf 1 steht, ist der Pin als Ausgang programmiert. Der Wert, der ins Datenregister PT*Xn* geschrieben wird, bestimmt dann den Ausgangspegel. Wird RDR*Xn* auf 1 gesetzt, wird der Treiberstrom von maximal 10 mA auf 2 mA reduziert. Dies reduziert den Stromverbrauch und elektromagnetische Abstrahlung, macht den Ausgang aber auch langsamer. Der Wert in PTI*X* ergibt den gleichen Wert wie in PT*X*, wenn der Ausgang nicht kurzgeschlossen ist.

Wird DDRXb auf 0 gesetzt, ist der Pin als Eingang programmiert. Der Wert am Anschluss kann über PTI*Xn* oder PT*Xn* gelesen werden; man kann auf PT*X* schreiben, aber beim Zurücklesen wird nicht der Registerwert, sondern der Wert am Anschluss zurückgegeben.

Wenn der Anschluss als Eingang konfiguriert ist, kann man ihn entweder mit einem Pulldown oder einem Pullup versehen. Um Pullup oder Pulldown zu aktivieren, muss man Register PER*Xn* mit einer 1 program-

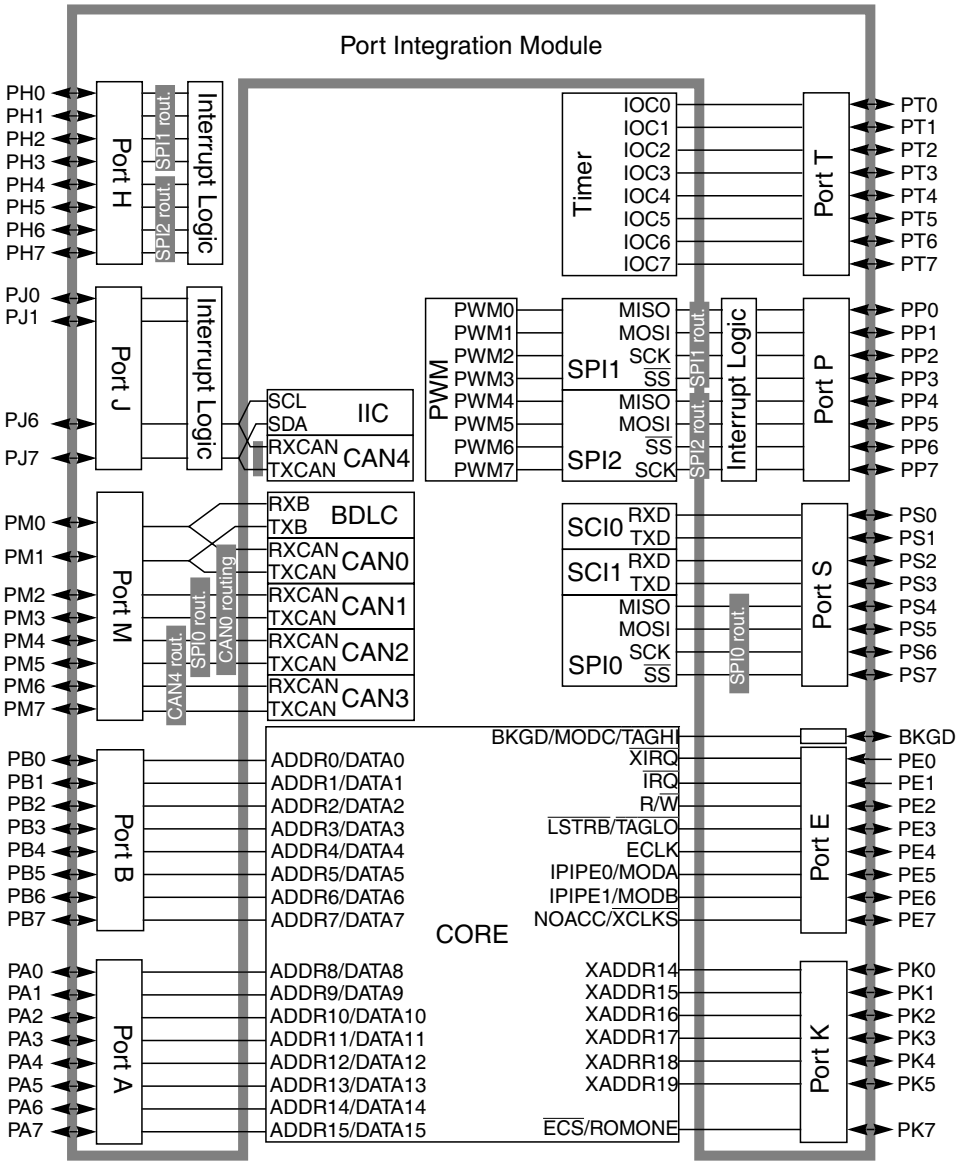


Abbildung 3-1: Überblick über die Ports des 68HCS12 9DP256

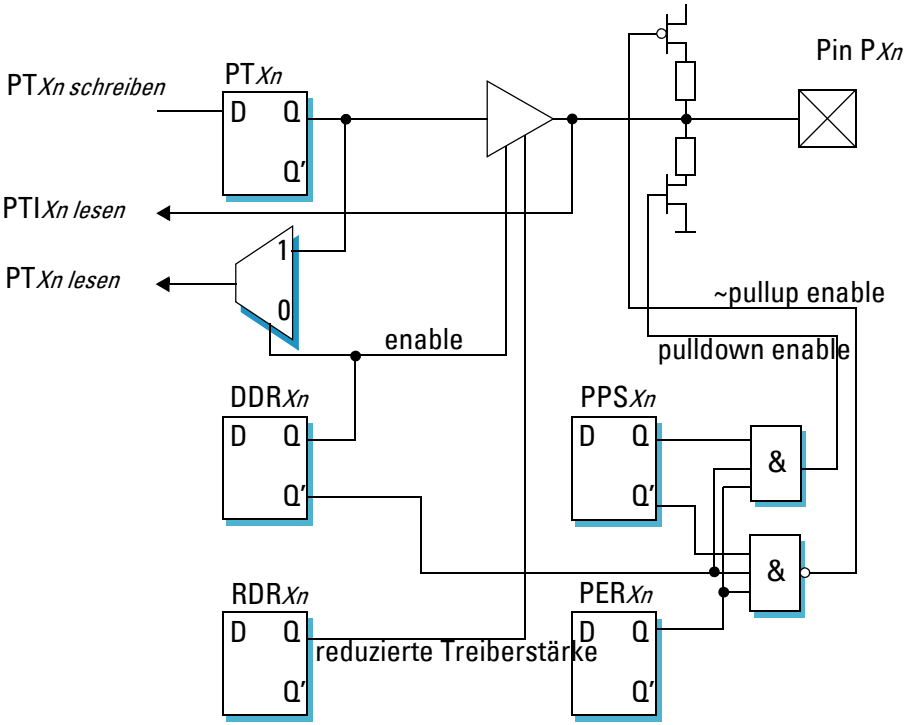


Abbildung 3-2: Schematische Darstellung eines General Purpose I/O-Pins

mieren. In diesem Fall versieht eine 1 in Register PPS*Xn* den Eingang mit einem Pulldown von ca. 130 µA, und eine 0 mit einem Pullup.

- Unbeschaltete Anschlüsse immer als Ausgang programmieren
- Beschaltete Anschlüsse (mit Pullup oder Pulldown) als Eingang programmieren
- Nie Eingänge einfach offen lassen (Latch-Up-Effekt, Prozessor wird deaktiviert, zieht viel Strom).

Man kann die Anschlüsse auch unabhängig voneinander verwenden, d.h. einen Teil als Eingang und einen anderen Teil als Ausgang programmieren. Dazu eignen sich besonders die Bit-Manipulationsbefehle. Wollten wir z.B. Port H Pin 2 als Eingang ohne Pullup oder Pulldown verwenden, könnten wir so vorgehen:

```
1    DDRH = DDRH & ~0x04
2    PERH = PERH & ~0x04
```

Wir könnten den Wert am Anschluss lesen und als Bedingung verwenden:

```
3       brsetPTIH,#4,branch1
```

Wollten wir den Pin3 des Port H als Ausgangspin mit reduziertem Treiberstrom betreiben, könnten wir so vorgehen:

```
4       bsetDDRH,#8
    bsetRDRH,#8
```

Wir würden den Anschluss auf 1 setzen mit

```
5       bsetPTH,#8
```

und auf 0 mit

```
6       bclrPTH,#8
```

Jeder Pin kann als Tristate-Pin verwendet werden, indem man ihn über Register DDRH entweder hochohmig, d.h. als Eingang konfiguriert, oder als Ausgang.

3.2.1 Mehrfach verwendete Anschlüsse

Einige Anschlüsse sind neben ihrer allgemeinen Funktion als Vielzweck-Anschlüsse mit speziellen Peripheriemodulen assoziiert, wie z.B. Analog-Digital-Wandlern, seriellen Schnittstellen oder Zeitgebern. Sobald diese speziellen Module aktiviert sind, überschreiben sie die Funktion des allgemein verwendbaren Anschlusses wie er weiter oben beschrieben worden ist. Einige Funktionen sind allerdings immer noch möglich, wie in Abbildung 3-3 gezeigt.

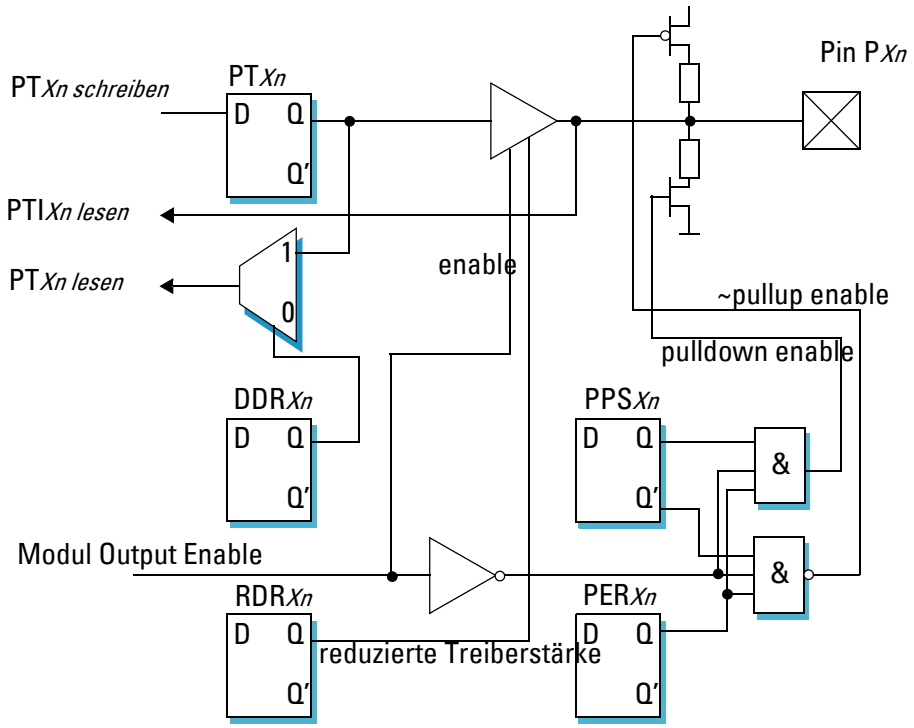


Abbildung 3-3: Überschriebene Funktion eines General Purpose I/O-Pins

Das Modul steuert die Richtung des Anschlusses (Ein- oder Ausgabe) und die Treiberstärke. Allerdings lässt sich die Treiberstärke bzw. die Pulldown- bzw. Pullup-Konfiguration weiterhin separat einstellen. Den Wert am Anschlusspin kann man immer über PTI*X* lesen. Das PT*X*-Register hat keine Wirkung, aber man kann darein schreiben, und wenn das DDR*X*-Bit gesetzt ist, auch daraus lesen.

3.2.2 Kernmodul-Ports

Die Ports A, B, E und K sind Teil des sogenannten Kernmoduls (CPU Core-Moduls), d.h. es gibt sie in allen Ausführungen dieses Prozessors. Diese Ports haben eine einfachere Struktur. Es gibt keine Pulldown-Möglichkeit, und die Pullups können nur gemeinsam für den gesamten Port über das PUCR-Register konfiguriert

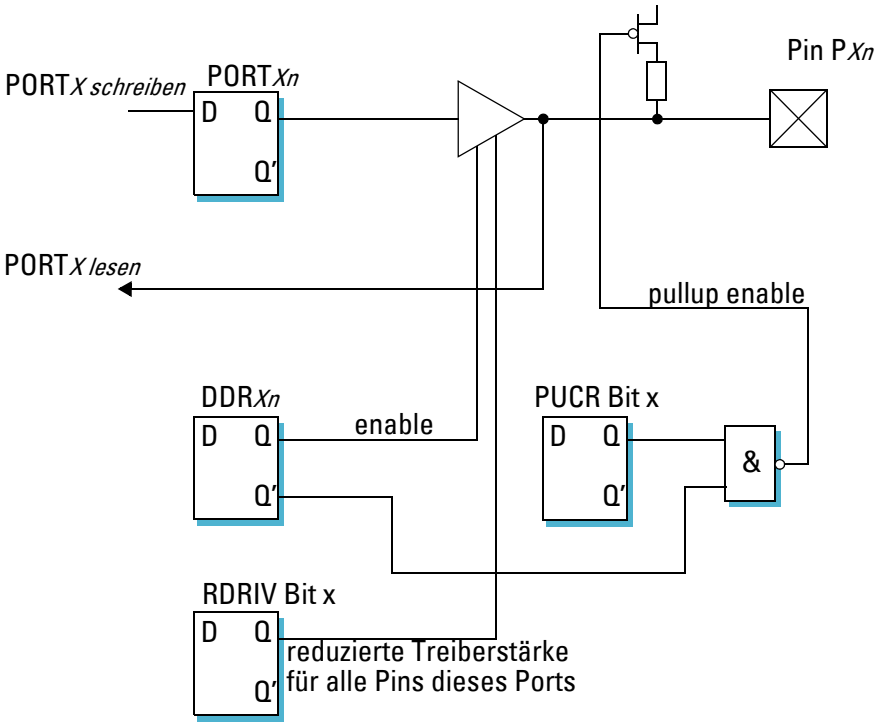


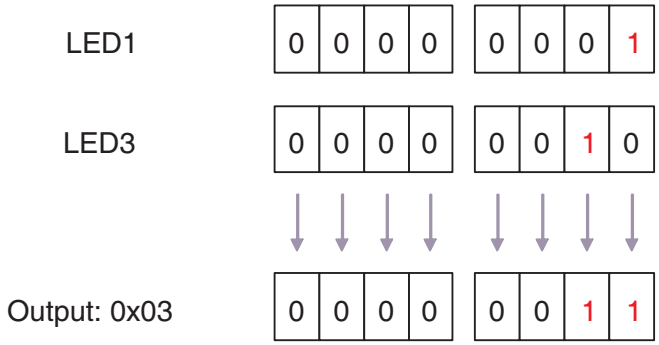
Abbildung 3-4: Ports des Kernmoduls (A, B, E und K)

werden. Die Treiberstärke kann auch nur insgesamt für den gesamten Port über das RDRIV-Register eingestellt werden. Ports E und K haben eine spezielle Bedeutung und sollten nicht für allgemeine Ein-Ausgabezwecke verwendet werden.

3.2.3 Bit-Manipulationen in C

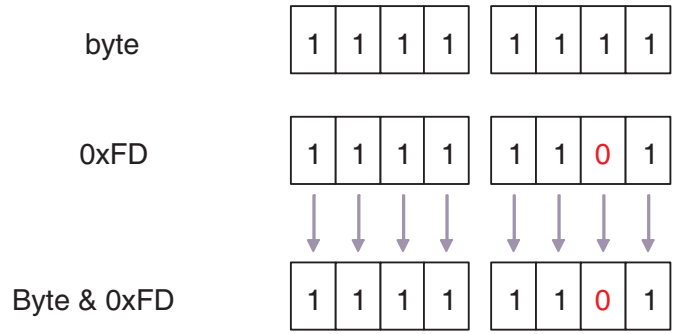
Bits setzen (die LED1, LED3 bezeichnet man als „*Maske*“):

```
enum flags {LED1 = 0x01, LED2 = 0x02, LED3 = 0x03, LED4 = 0x04};
output(LED_PORT , LED1 | LED3);
```



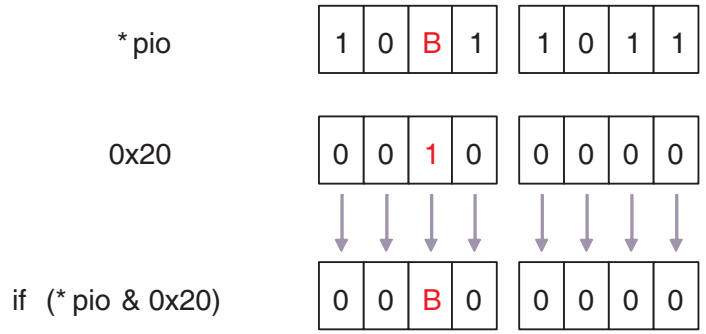
Bits löschen (die 0xFD nennt man „*Maske*“):

```
unsigned  char  byte  0xFF;
byte  &= 0xFD; // 1111 1101: Lösche Bit 1
```



Bits testen (die 0x20 bezeichnet man als „Maske“):

```
unsigned  char  *pio  =  reinterpret_cast<unsigned   char*>(0x8000);
if  (*pio  &  0x20)          // teste, ob Bit 5 gesetzt ist
```



3.3 Zeitgeber (Timer)

In vielen Anwendungen benötigen Rechner ein Konzept von Zeit. Dies gilt insbesondere für Echtzeitsysteme, aber auch z.B. für Computerspiele, die in einem bestimmten Tempo ablaufen sollen. Aus diesem Grund besitzen die meisten Rechner Zeitgeber, die so konfiguriert werden können, dass bei Ablauf einer Zeit ein Interrupt erzeugt wird. Kontroller für eingebettete Anwendungen besitzen z.T. sehr umfangreiche Zeitgebermodule, um z.B. Aktoren wie Servomotoren oder Magnetventile zu steuern, und Zeiten und Impulsfrequenzen zu messen, wie es z.B. in vielen Anwendungen im Automobil erforderlich ist.

Der von uns eingesetzte Mikrocontroller besitzt ein sehr umfangreiches Zeitgebermodul, das hier nur z.T. besprochen werden kann. Wir können seine Funktionalität in vier Klassen aufteilen:

- Freilaufender Zähler
- Zeitmessung bei Eingangssignal (Input Capture)
- Erzeugung zeitabhängiger Ausgangssignale (Output Compare)
- Impulszähler

Der Kontroller besitzt einen durch einen Oszillator getriebenen 16-Bit-Zähler. Dieser Zähler kann für die Messung relativer Zeiten benutzt werden. Daneben stellt der Kontroller 8 Zeitgeber-Kanäle zur Verfügung, die verschieden konfiguriert werden können.

Im „Input Capture“-Modus wird der Wert des 16-Bit-Zählers in ein Register übernommen, wenn am zugeordneten Eingangspin ein Signalübergang detektiert worden ist. Damit kann man z.B. Frequenzen oder Periodendauern eines externen Signals messen.

Ein Kanal kann auch im „Output Compare“-Modus betrieben werden. Hier wird der zugeordnete Ausgangspin auf einen bestimmten Pegel gesetzt oder verändert, sobald der 16-Bit-Zähler den im zugeordneten Vergleichsregister eingestellten Wert erreicht hat. Damit kann man zeitlich genau definierte Ausgangssignale z.B. für die Steuerung von Servomotoren oder auch die Ansteuerung eines kleines Lautsprechers erzeugen. Allerdings bietet der Kontroller noch ein eigenständiges Modul für die Erzeugung von pulsweitenmodulierten Signalen an.

Es steht ein 16-Bit Impulszähler zur Verfügung, mit dessen Hilfe man Eingangsimpulse summieren kann. Zusammen mit dem Zeitgeber kann man so z.B. Frequenzmessungen vornehmen oder auch sehr lange Zeiten messen.

Dem Zeitgebermodul ist der Port T zugeordnet. Jedem Zeitgeberkanal ist einer der 8 Anschlüsse des Ports T zugeordnet.

Der Haupttakt für das Zeitgebermodul leitet sich aus dem Prozessor-Bustakt ab. Dieser ist bei unserem Rechner in der Betriebsart als Evaluation Board (EVB) auf 24 MHz eingestellt. Im Boot-Mode muss der Anwendungsprogrammierer die Frequenz selbst einstellen; der voreingestellte Wert ist deutlich niedriger als 24 MHz.

3.3.1 Freilaufender Zähler

Der freilaufende Zähler des Zeitgebermoduls hat eine zentrale Bedeutung. Die acht Zeitgeberkanäle benutzen den Wert dieses Zähler zur Realisierung ihrer Funktionen. Der eigentliche Zähler ist über das Register TCNT erreichbar, er ist nur lesbar. Der Zähler wird nicht direkt vom Bustakt getaktet, sondern über einen Vorteiler (Prescaler). Dieser ist nach einem Reset auf einen Teilfaktor von 1 eingestellt. Über das Register TSCR2, Bits 0 bis 2 (PR2, PR1, PR0) lassen sich Teilfaktoren von 2^n einstellen, wobei n der Wert der aus PR2, PR1 und PR0 gebildeten Binärzahl ist. Der maximale Teilfaktor ist also 128, der minimale 1.

Um eine Zeitdifferenz zu messen, speichern wir zunächst den aktuellen Wert von TCNT als Anfangszeit. Später laden wir wieder den dann aktuellen Wert von TCNT und ziehen davon den Anfangswert ab:

```
7       movwTCNT, startZeit; startZeit ist eine Wort-Variable
8       ; hier kann jetzt etwas passieren, von dem wir die Zeit messen wollen
9       ldd TCNT      ; hier laden wir die neue Zeit
    subdstartZeit  ; in D steht jetzt die Zeitdifferenz
```

Der absolute Wert von TCNT ist nicht relevant, und es spielt auch keine Rolle, wenn der Zähler während der Messung überläuft. Die einzige Einschränkung ist, dass die Differenz nicht größer als 65535 Zähleinheiten betragen darf.

Immer wenn der Zähler überläuft, wird das TOF-Status-Bit gesetzt. Man kann das Bit dadurch zurücksetzen, dass man eine 1 in das Register schreibt. Ist das TFFCA-Steuer-Bit gesetzt, wird das TOF durch Lesen des

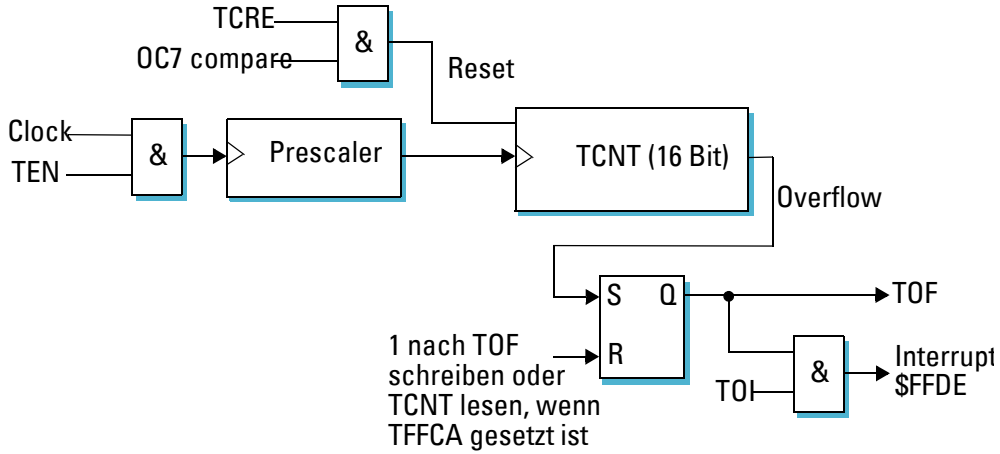


Abbildung 3-5: Konfiguration des freilaufenden Zählers

TCNT automatisch zurückgesetzt. In diesem Fall funktioniert das Rücksetzen durch Schreiben einer 1 nach TOF nicht mehr.

Tabelle 3-1: Timer Count Control und Status-Bits

Timer Count Control und Status-Bits								
Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TSCR1	TEN	TSWAI	TSFRZ	TFFCA	0	0	0	0
TSCR2	TOI	0	0	0	TCRE	PR2	PR1	PR0
TFLG2	TOF	0	0	0	0	0	0	0

Durch Setzen des TOI-Bits kann man dafür sorgen, dass ein Zählerüberlauf zu einem Interrupt führt. Auf diese Weise kann man den Hardwarezähler um einen Softwarezähler von praktisch beliebiger Länge erweitern. Beispiel für die Erweiterung auf 32 Bit:

```
10  timerInterruptServiceRoutine:
11      ldd timerExtension; liegt irgendwo im Speicher (ds.w 1)
    addd#1
```

```
    std timerExtension
    movb#$80,TFLG2; Interrupt-Flag zurücksetzen
    rti
```

Probleme kann es hier beim Lesen des Zählers geben, da sich der Wert während des Lesens ändern kann. Das wird sicher nur selten passieren, aber es kann passieren:

```
12      movwtimerExtension, startZeit
    movw TCNT, startZeit+2
```

Der `movw`-Befehl benötigt 6 Bustaktzyklen für seine Ausführung. In dieser Zeit kann es schon zu einem Zählerüberlauf gekommen sein, so dass z.B. nach der Ausführung des ersten Befehls die Interrupt-Service-Routine angesprungen wird. Sie inkrementiert den Wert in `timerExtension`, während TCNT wieder bei 0 beginnt. Wenn wir aus der Interrupt-Service-Routine zurückkehren und TCNT gelesen haben, steht in `startZeit` und `startZeit+2` ein inkonsistenter Wert (wir sind in der Zeit zurückgefallen).

3.3.2 Zeitmessung von Eingangssignalen (Input Capture)

Diese Betriebsart behandeln wir zur Zeit nicht.

3.3.3 Zeitabhängige Ausgangssignale (Output Compare)

Mit Hilfe der „Output Compare“-Funktionalität lassen sich Einzelimpulse oder Impulsfolgen erzeugen. Damit ein Kanal in dieser Betriebsart arbeiten kann, muss der Hauptzeitgeber, der freilaufende Zeitzähler aktiviert sein (TEN=1) und das entsprechende Bit IOS*n* im Register TIOS muss von seinem voreingestellten Wert von 0 auf 1 gesetzt werden.

Dadurch wird der Wert von TCNT kontinuierlich mit dem Wert des Registers TC*n* verglichen, und wenn die beiden Werte übereinstimmen, wir das Flag-Bit C*n*F im Register TFLG1 gesetzt. Je nach Einstellung der

Bits OM*n* und OL*n* in den Registern TCTL1 bzw. TCTL2 wird in diesem Fall der Ausgangspin PT*n* gesetzt, zurückgesetzt, oder sein Wert geändert.

Tabelle 3-2: Timer Output Compare Control und Status-Bits

Timer Output Compare Control und Status-Bits								
Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TIOS	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0
CFORC	FOC/	FOC6	FOC5	FOC4	FOC3	FOC2	FOC1	FOC0
TSCR1	TEN	TSWAI	TSFRZ	TFFCA	0	0	0	0
TSCR2	TOI	0	0	0	0	0	0	0
TCTL1	OM7	OL7	OM6	OL6	OM5	OL5	OM4	OL4
TCTL2	OM3	OL3	OM2	OL2	OM1	OL1	OM0	OL0
TIE	C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I
TFLG1	C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F

Mit Hilfe der Bits in TIE lässt sich für jeden der acht Kanäle konfigurieren, ob bei Gleichstand der TCNT und TC*n*-Register ein Interrupt erzeugt wird.

Das folgende kleine Programm erzeugt eine Rechteckschwingung mit einer Frequenz von 10 kHz am Anschluss PT4. Der Kanal 4 des Zeitgebermoduls wird hier im Polling-Betrieb verwendet. Eine Frequenz von 10 kHz entspricht einer Periodendauer von 100 µs; der Pegel muss aber alle 50 µs geändert werden, damit sich eine symmetrische Rechteckschwingung ergibt:

```
13   ...
14       bset TSCR1,#$90; setze TEN und TFFCA-Bits
     bset TIOS, #$10; setze IOS4 (Kanal 4 im Output Compare-Modus)
     bsetTCTL1, #1; setze OL4 (ändere Ausgang PT4 bei Gleichstand)
     ldd TCNT    ; lade momentanen Zählerstand
     addd#50*24 ; addiere dazu 50 Mikrosekunden
     std TC4     ; speichere diesen Zukunftswert im Vergleichsregister
loop1:
     brclrTFLG1, #$10, loop1; warte, bis C4F-Flag gesetzt ist
```

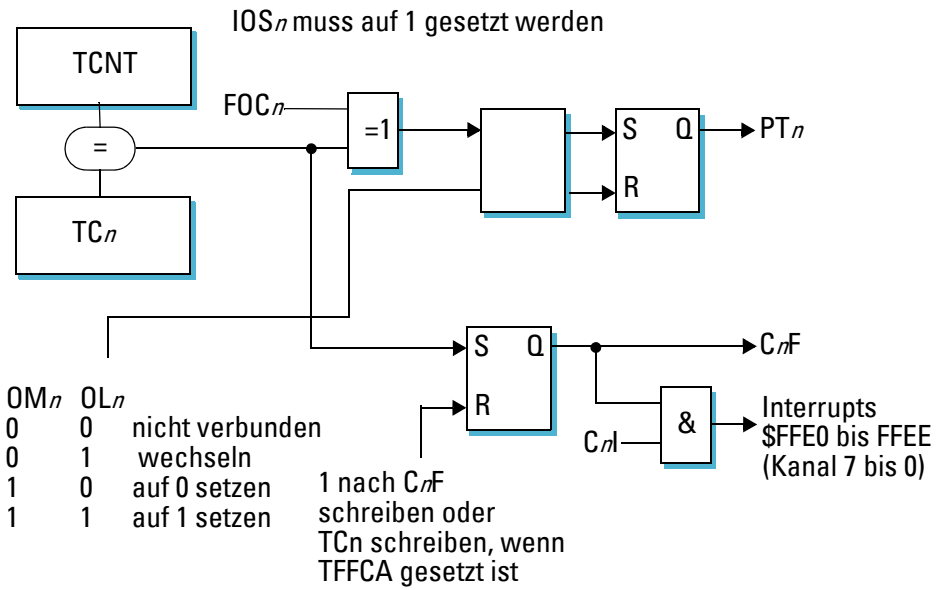


Abbildung 3-6: Zeitgeber im Output Compare-Modus

```
addd#50*24 ; berechne nächsten Zeitpunkt
std TC4    ; setze neue Zeit und setze Flag zurück
bra loop1
```

Die ersten drei Befehle initialisieren das Zählermodul. Die Konfiguration ist so gewählt, dass der Zustand des Anschlusses bei jedem erfolgreichen Vergleich wechselt, also von 1 auf 0, und danach von 0 auf 1 usw.

Die auf diese Weise erzielbare maximale Pulslänge erreichen wir, wenn wir zum momentanen Zählerstand 65535 dazu addieren. Die minimale Periode ist dadurch beschränkt, dass die vier Befehle innerhalb der Abfrageschleife in der halben Periode abgearbeitet werden können müssen.

3.4 Analog-Digitalwandler

Viele Anwendungen erfordern eine Anbindung von Rechnern an analoge Sensoren, z.B. Beschleunigungsaufnehmer, Temperaturfühler oder Drucksensoren. Damit solche Werte im Rechner verarbeitet werden können, müssen sie zuerst digitalisiert werden.

Der von uns verwendete Rechner besitzt zwei Analog-Digital-Konverter mit jeweils 10 Bit Auflösung, die nach dem Verfahren der sukzessiven Approximation arbeiten. Jeder der beiden A/D-Wandler kann über einen Analog-Multiplexer auf einen von jeweils acht Eingängen geschaltet werden. Der Wandler ist relativ schnell; er wird mit maximal 2 MHz getaktet und benötigt 14 Taktzyklen für eine komplette Wandlung. Das entspricht einer Wandlungszeit von 7 µs. Betreibt man den Wandler im 8-Bit-Modus, kann die Anzahl der Wandlungszyklen um 2 reduziert werden, d.h. die minimale Wandlungszeit beträgt dann 6 µs.

Der A/D-Wandler benötigt zwei Referenzspannungen, VRH und VRL. Die zu messenden Spannungen müssen zwischen diesen beiden Werten liegen. Die Auflösung des A/D-Wandlers beträgt damit (VRH-VRL)/1024 Volt. Bei einer Differenz von 5 Volt für (VRH-VRL) beträgt die Auflösung damit ca. 5 mV.

3.4.1 Initialisierung

Der Mikrocontroller besitzt zwei A/D-Wandler. Die Steuerungs- und Statusregister des ersten Wandlers heißen ATD0xxx und liegen im Speicherraum ab Adresse $80. Die Datenregister heißen ADR0xxx bzw. PORTAD0, wenn die Anschlüsse als Digitaleingänge verwendet werden.

Der Wandler muss vor seiner Verwendung über die Steuerungsregister 2 bis 4 konfiguriert werden. Die Konfigurationsphase wird durch Schreiben in das Steuerungsregister 5 abgeschlossen; damit wird gleichzeitig eine Wandlung gestartet.

Das ADPU-Bit aktiviert das A/D-Wandlermodul. In der Voreinstellung ist das Modul abgeschaltet, da es einen relativ hohen Stromverbrauch hat. Der A/D-Wandler benötigt 10 µs um betriebsbereit zu sein, nachdem das ADPU-Bit gesetzt worden ist. Alle anderen Bits in ATDxCTL3 können normalerweise auf 0 gesetzt werden. Wird Bit ASCIE gesetzt, wird am Ende einer Wandlung ein Interrupt ausgelöst und das Interrupt-Flag ASCIF wird gesetzt. Durch Schreiben in ATDxCTL5 wird das Flag zurückgesetzt und eine neue Wandlungssequenz wird gestartet.

Die Bits in ATDxCTL3 und ATDxCTL4 müssen richtig gesetzt werden, damit der Wandler wie gewünscht arbeitet. Bits S8C bis S1C bestimmen die Anzahl der Wandlungen pro Befehl. Man darf 1 bis 8 Wandlungen einstellen; andere Werte bedeuten 8 Wandlungen. Die Ergebnisse werden in den Ergebnisregister ADRx0 bis ADRx7 abgelegt, jeweils beginnend mit ADRx0. Ist das FIFO-Bit gesetzt, werden weitere Ergebnisse angehängt. Bei fünf Wandlungen pro Befehl liegen der Ergebnisse also in ADRx0, ADRx1, ADRx2, ADRx3 und ADRx4 für den ersten Wandlungsbefehl, und in ADRx5, ADRx6, ADRx7, ADRx0, ADRx1 für den zweiten Wandlungsbefehl.

Tabelle 3-3: Analog-Digitalwandler Control- und Status-Bits

Analog-Digitalwandler Control- und Status-Bits								
Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ATD0CTL2	ADPU	AFFC	AWAI	ETRIGLE	ETRIGP	ETRIGE	ASCIE	ASCIF
ATD0CTL3	0	S8C	S4C	S2C	S1C	FIFO	FRZ1	FRZ0
ATD0CTL4	SRES8	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0
ATD0CTL5	DJM	DSGN	SCAN	MULT	0	CC	CB	CA
ATD0STAT0	SCF	0	ETORF	FIFOR	0	CC2	CC1	CC0
ATD0STAT1	CCF7	CCF6	CCF5	CCF4	CCF3	CCF2	CCF1	CCF0

Die Bits FRZ1 und FRZ0 bestimmen, ob im Debug-Modus die Wandlungen weitergehen, wenn der Prozessor auf einen Breakpoint gelaufen ist.

Der A/D-Wandler darf höchstens mit 2 MHz und muss mindestens mit 500 kHz getaktet werden. Die Taktfrequenz ergibt sich aus dem Bustakt geteilt durch einen Vorteiler. Der Vorteiler wird mit Hilfe der Bits PRS0 bis PRS4 eingestellt. Um z.B. bei einem Bustakt von 24 MHz den maximalen A/D-Wandler-Takt zu erzielen, müsste der Vorteiler auf einen Wert von 12 eingestellt werden. Der Teiler ergibt sich aus der aus PRS0 bis

PRS4 gebildeten Dualzahl mal 2. Ein Vorteilerfaktor von 12 wäre also 0010 (PRS4...PRS0), was auch die Voreinstellung ist.

Tabelle 3-4: Analog-Digitalwandler Control- und Status-Bits, konkrete Werte

Analog-Digitalwandler Control- und Status-Bits								
Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ATD0CTL2	1	0	0	0	0	0	0	0
ATD0CTL3	0	0	0	0	1	0	0	0
ATD0CTL4	0	1	0	0	0	1	0	1

Mit SRES8 gesetzt arbeitet der Wandler als 8-Bit-Wandler. SMP1 und SMP0 bestimmen die Anzahl der Taktzyklen für den letzten Abtastzyklus; sie sollten auf ihrem voreingestellten Wert von 10 belassen werden.

Damit sieht eine typische Konfiguration wie in Tabelle 3-4 gezeigt aus.

3.4.2 Kanalselektion

Jeder Analog-Digitalwandler kann mit einem von acht Eingängen verbunden werden. Dazu dienen die Bits CA bis CC im Register ATDxCTL5. Ist das DJM-Bit gesetzt, werden die Wandlungsergebnisse als Integer im Wertebereich 0 bis 1023 zur Verfügung gestellt. Steht dieses Bit auf 0, wird das Ergebnis als binäre Rationalzahl dargestellt (0,1111...). Das erlaubt es, die Auswertung unabhängig davon zu halten, ob man den Wandler mit 8- oder 10-Bit Auflösung betreibt. Außerdem kann man mit Bit DSGN auch eine vorzeichenbehaftete Darstellung einstellen.

Wird das Bit MULT gesetzt, wird automatisch für jeden Analogeingang eine Wandlung vorgenommen, beginnend bei dem mit Ca bis CC eingestellten Eingang, modulo 8.

Wird das Bit SCAN gesetzt, arbeitet der Wandler kontinuierlich, ohne dass man die Wandlungen anstoßen müsste.

Eine Wandlungssequenz wird ansonsten begonnen, indem man in das Register ATDxCTL5 schreibt. Sobald ein Ergebnis in ein Register ADRxx gespeichert worden ist, wird das entsprechende Flag CCFx gesetzt. Die CCx-Bits in Register ATDxSTAT0 zeigen an, wohin das nächste Ergebnis gespeichert werden wird.

Das Bit AFFC in ATD0CTL2 bestimmt den „Fast Flag Clear All“-Modus des A/D-Wandlers. Ist es auf 0 gestellt, wird das CCFx-Bit durch Lesen von ATD0STAT1 und anschließendes Lesen des zugehörigen Datenregisters zurückgesetzt. Steht AFFC auf 1, genügt das Lesen des jeweiligen Datenregisters, um das SCF-Bit und das zugehörige CCFx-Bit zurückzusetzen.

3.5 Digital-Analogwandler

3.6 Pulsweitenmodulator

Pulsweitenmodulation ist eine verbreitete Technik, um digitalisierte Datenwerte über eine einzige Leitung ohne zusätzlichen Takt zu übertragen. Pulsweitenmodulation wird z.B. für die Ansteuerung von Servomotoren, in der Audio-Datenübertragung (Mobiltelefonen) zur Erzeugung von Tönen und Geräuschen sowie zur einfachen Digital-Analogwandlung benutzt.

Bei der Pulsweitenmodulation wird eine kontinuierliche Folge von Rechteckimpulsen mit einer konstanten Frequenz erzeugt. Variiert wird die Zeit zwischen der steigenden und fallenden Flanke eines Impulses, also die „Weite“ des Impulses. Die Weite des Impulses repräsentiert damit einen analogen Wert zwischen 0% einer Taktperiode und 100% einer Taktperiode.

Für die Erzeugung pulsweitenmodulierter Signale benötigt man eine Zähler und zwei Register; in einem Register wird die Pulsweite eingestellt, in dem anderen die Taktperiode. Zu Beginn einer Periode wird der Zähler auf 0 gesetzt und der Ausgang auf 1. Der Zähler wird durch einen Taktgeber erhöht. Sobald der Wert im Zähler dem Wert im ersten Register entspricht, wird der Ausgang auf 0 gesetzt. Sobald der Wert im Zähler den Wert im zweiten Register erreicht, wird er auf 0 gesetzt und ein neuer Zyklus beginnt.

Der Wert im ersten Register muss immer kleiner sein als der Wert im zweiten Register. Die Pulsweite wird dadurch moduliert, dass man den Wert des ersten Registers verändert.

Bei der Pulsdichtemodulation geht man ähnlich vor; hier wird allerdings die Pulsweite konstant gehalten und die Taktperiode variiert.

Unser im Labor verwendeter Rechner besitzt acht PWM-Kanäle mit jeweils 8-Bit-Zählern und Registern. Diese können aber zu insgesamt vier Kanälen mit 16-Bit Zählern und Registern zusammengefasst werden.

Die PWM-Kanäle sind dem Port P zugeordnet. Dieser Port wird auch vom Serial Peripheral Interface (SPI) verwendet. Jeder Pin kann entweder einem PWM-Kanal oder dem SPI zugeordnet werden. Ist keins von beiden der Fall, steht er als allgemeiner Port-Anschluss zur Verfügung.

Die PWM-Generatoren werden vom Systemtakt getrieben, allerdings nicht direkt, sondern über konfigurierbare Vorteiler (Prescaler). Es gibt zwei Vorteiler-Kanäle, die unterschiedlich eingestellt werden können. Der Vorteiler A wird von den PWM-Kanälen 0, 1 4 und 5 benutzt, während der Vorteiler B von den PWM-Kanälen 2,3,6 und 7 benutzt wird.

Die Teile bestehen aus zwei hintereinandergeschalteten Einheiten. Die erste Einheit teilt durch Zweierpotenzen von 2^0 bis 2^7. Die zweite Teilerstufe der Kaskade erlaubt eine Teilung durch N, mit N =1 bis 256. Der zweite Teiler ist optional. Wird er nicht benutzt, ist der Teilfaktor 2^M, mit M = 0 bis 7. Wird er benutzt, ist der Teilfaktor 2^{M+1}*N, es findet also noch zusätzlich eine Teilung durch 2 statt.

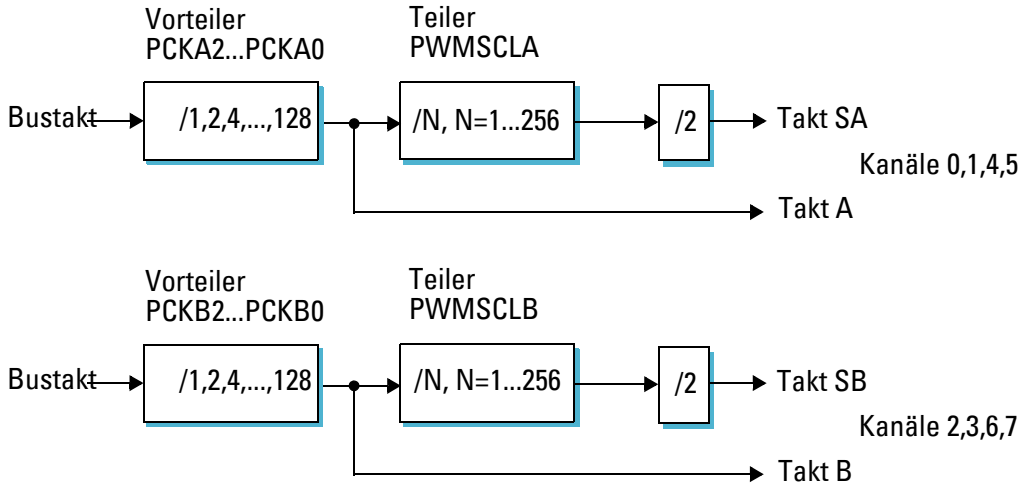


Abbildung 3-7: Konfiguration der PWM-Vorteiler

Ein PWM-Kanal wird durch Setzen des entsprechenden PWME*n*-Bits aktiviert. Die Polarität des Ausgangssignals wird durch das entsprechende PPOL*n*-Bit bestimmt. Ist es auf 1 gesetzt, werden positive Pulse erzeugt (logisch 1). Jeder Kanal kann mit oder ohne zweiten Vorteiler betrieben werden. Soll der Teiler verwendet werden, d.h. der PWM-Kanal durch die Takte SA bzw. SB getrieben werden, muss das entsprechende PCLK*n*-Bit auf 1 gesetzt werden.

Tabelle 3-5: Pulsweitenmodulator Control- und Status-Bits

Pulsweitenmodulator Control- und Status-Bits								
Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PWME	PWME7	PWME6	PWME5	PWME4	PWME3	PWME2	PWME1	PWME0
PWMPOL	PPOL7	PPOL6	PPOL5	PPOL4	PPOL3	PPOL2	PPOL1	PPOL0
PWMCAE	CAE7	CAE6	CAE5	CAE4	CAE3	CAE2	CAE1	CAE0
PWMCLK	PCLK7	PCLK6	PCLK5	PCLK4	PCLK3	PCLK2	PCLK1	PCLK0
PWMPRCLK	0	PCKB2	PCKB1	PCKB0	0	PCKA2	PCKA1	PCKA0
PWMCTL	CON67	CON45	CON23	CON01	PSWAI	PFRZ	0	0
PWMSCLA	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PWMSCLB	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Jeder PWM-Kanal hat ein Zählregister PWMCNT*n* und zwei Vergleichsregister PWMDTY*n* und PWMPER*n*. Die Pulsweite (duty cycle) wird über PWMDTY*n* eingestellt, die Periode über PWMPER*n*. Schreiben auf PWMCNT*n* setzt diesen Zähler zurück auf 0.

Mit Hilfe des PWMCAE-Registers kann man einstellen, dass Pulse zentriert und nicht links ausgerichtet werden. In diesem Fall sind die Pulsweiten und Taktperioden doppelt so hoch. Ein Periode startet immer in der Mitte eines Pulses.

Angenommen, wir wollten 1 µs weite Impulse mit einer Taktperiode von 10 µs erzeugen. Die Bustaktfrequenz beträgt 24 MHz. Um eine Auflösung von 1 µs zu erhalten, benutzen wir die kaskadierten Teiler mit M=0 und N=12. Damit erhält das Taktperiodenregister einen Wert von 10 (10 mal 1 µs) und das Duty Cy-

cle-Register einen Wert von 1. Der folgende Programmausschnitt konfiguriert den Kanal 0 mit Pin 0 an Port P:

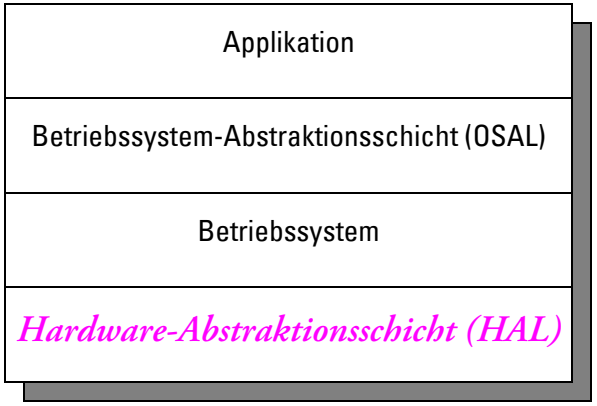
```
movb  #12, PWMCSCLA     ; Taktrate von 1 µs
movb  #1, PWMCLK        ; nutze Takt SA
movb  #1, PWMPOL        ; positive Pulse
movb  #1, PWMDTY0       ; Pulsweite 1 * 1 µs
movb  #10, PWMPER0      ; Periode ist 10*1 µs
movb  #1, PWME          ; akiviere Kanal 0
```

3.7 Software-Strukturierung

Es ist vorteilhaft, Softwaresysteme geschichtet aufzubauen:

- bessere *Übersicht*
- bessere *Testbarkeit*
- bessere *Portierbarkeit*

Typisches Software-Schichtenmodell:



- Betriebssystem ist optional
- HAL beinhalten bei Einsatz von Betriebssystem oft „*Treiber*“ (driver)
- Betriebssystem-Abstraktionsschicht selten verwendet (Beispiel OSAL von Rhapsody)

Wir schauen uns zunächst nur Applikation (die eigentliche Anwendung) und HAL für ein einfaches Beispielsystem an. Diesen realisieren wir als *Foreground/Background-System*.

3.8 Erster Laborversuch: Foreground/Background-System

Praktisch jeder Rechner arbeitet die folgenden Schritte immer wieder ab, bis er abgeschaltet wird:

1. *Befehl* aus Speicherzelle *holen*, auf die der Programmzähler zeigt
2. *Programmzähler* auf nächsten Befehl *stellen*
3. *Befehl dekodieren*

4. *Befehl ausführen*

Wenn der Programmzähler auf seinem maximalen Wert angekommen ist, fängt er wieder beim minimalen Wert von vorne an (z.B. kommt bei unserem Rechner im Labor nach $FFFF dann $0000). Das ist selten sinnvoll (bei unserem Laborrechner liegen bei $0000 gar keine Programme). Deshalb sorgt der Programmierer dafür, dass er die Schleife unter Kontrolle hat:

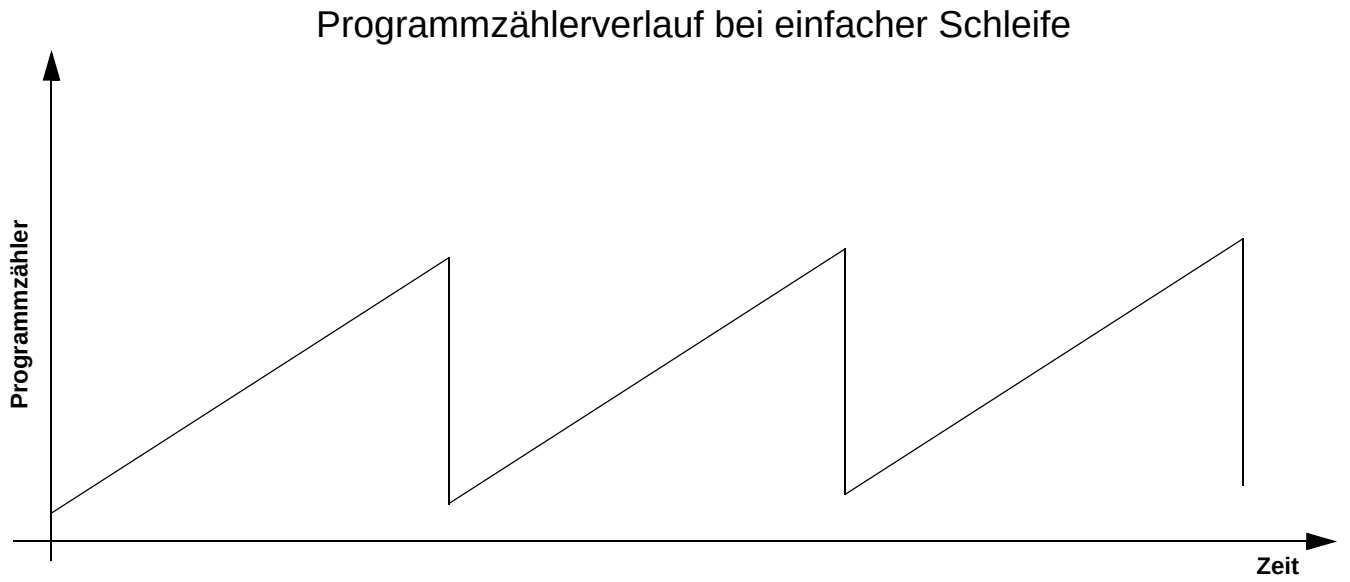
```
int main() {
  /* do here what needs to be done once ... */
  for (;;) {
     /* do here what needs to be done all the time...*/
  }
}
```

Damit könnte man schon die Uhr für unseren Tea-Timer realisieren (der Programmzähler wird von einem quartzgesteuerten Takt erhöht), wenn in der Schleife *immer die gleiche Zeit* verbraucht würde.

Leider ist das praktisch nicht zu realisieren: beim Überlauf der Sekunden auf eine Minute läuft in der Schleife etwas anderes ab als dazwischen, und schon ist die Uhr ungenau. Wir können also einen genauen Sekundenzähler nicht direkt an die Schleife binden.

Lösung: wir benutzen einen Hardwarezähler, und der sagt unserem Progamm mikrosekundengenau, wenn mal wieder eine bestimmte Zeit vergangen ist.

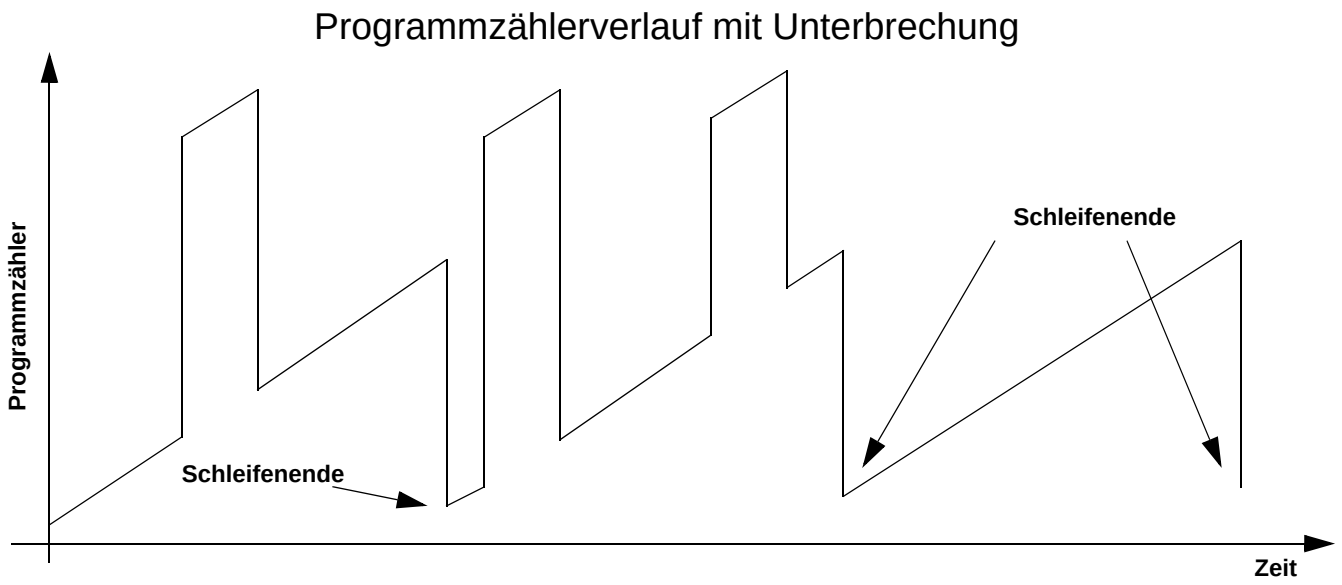
Wie sagt er es unserem Programm? Er unterbricht es kurz durch eine *Unterbrechungsroutine*!



- Programmzähler wird erhöht, bis Ende der Schleife erreicht ist
- Danach wird wieder beim Beginn der Schleife begonnen, usw.

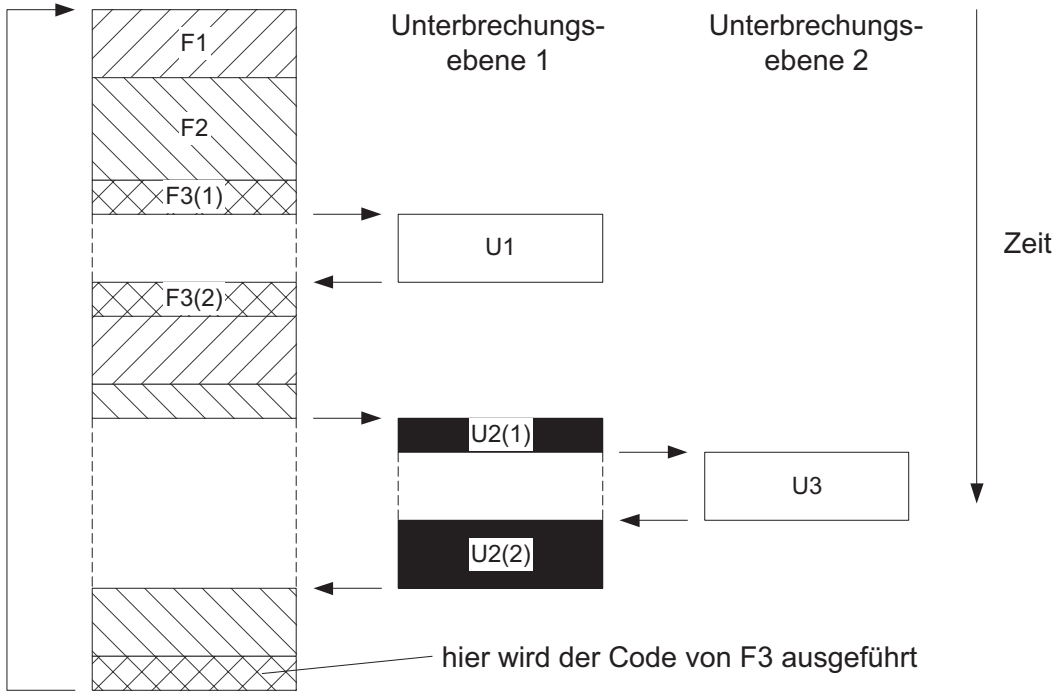
(In diesem simplen Beispiel gibt es keine Verzweigungen innerhalb der Schleife)

Die meisten Rechner bieten die Möglichkeit, aufgrund von *Hardwaresignalen* oder Ereignissen den Verlauf des Programmzählers zu ändern.



Eine Programmstruktur, die aus einer Hauptschleife mit Unterbrechungsroutinen besteht, nennt man *Foreground/Background-System*.
Es ist die einfachste Softwarestruktur für ein dediziertes System.

Andere Darstellung:



3.9 Unterbrechungsbehandlung

Noch einmal zur Wiederholung, ein Rechner arbeitet ein Programm normalerweise so ab:

1. *Befehl* aus Speicherzelle *holen*, auf die der Programmzähler zeigt
2. *Programmzähler* auf nächsten Befehl *stellen*
3. *Befehl dekodieren*
4. *Befehl ausführen*

Der Programmzähler wird mit jedem Befehl erhöht.

Der Wert kann bei Verzweigungen im Programm auch springen.
Die meisten Rechner können den Programmzähler noch über *Ausnahmen* (*engl.* exception) verstellen.
Es gibt drei Arten von Ausnahmen:

1. *Interrupts* (Unterbrechungen)
2. *Traps* (Fallen, Signale)
3. *Reset*

Interrupts sind asynchrone Ereignisse, die von Peripheriegeräten und Zeitgebern verursacht werden.
Traps sind durch Software oder die interne Rechnerhardware angestoßene Ereignisse

- Software Interrupts
- Hardwarefehler (Division durch 0, nicht ausgerichteter Speicherzugriff, etc.)

Ausnahmen *unterbrechen* den normalen *Programmablauf*.
Behandlung einer Ausnahme erfolgt in einem *Unterbrechungsprogramm* (Interrupt Service Routine)
Nach Behandlung Rückkehr zum unterbrochenen Programm oder auch nicht (fataler Fehler, Wechsel zu neuer Aufgabe).
Bei Reset nie Rückkehr zum unterbrochenen Programm (Rückkehradresse meist verloren).
Interrupt-Vektortabelle: bestimmte Region im Adressraum, in der für jede mögliche Interruptursache die Adresse der zugehörigen Interrupt-Behandlungsroutine abgelegt ist.

Tabelle 3-6: Interrupt-Vektortabelle (Auszug)

Vektor-adresse	Interruptquelle	CCR-Maske	Enable	HPRIO-Wert
$FFFE,$FFFF	Reset	-	-	-
$FFFA,$FFFB	COP failure reset	-	COP rate select	-
$FFF8,$FFF9	Unimplemented instruction trap	-		-

Tabelle 3-6: Interrupt-Vektortabelle (Auszug)

Vektor-adresse	Interruptquelle	CCR-Maske	Enable	HPRIO-Wert
$FFF6,$FFF7	SWI	-		-
$FFF4,$FFF5	XIRQ	X-Bit		-
$FFF2,$FFF3	IRQ	I-Bit	IRQCR(IRQEN)	$F2
$FFF0,$FFF1	Real Time Interrupt	I-Bit	CRGINT(RTIE)	$F0
$FFEE,$FFEF	Enhanced Capture Timer channel 0	I-Bit	TIE(C0I)	$EE
$FFEC,$FFED	Enhanced Capture Timer channel 1	I-Bit	TIE(C1I)	$EC
	Enhanced Capture Timer channel 4	I-Bit	TIE(C4I)	
	Enhanced Capture Timer channel 5	I-Bit	TIE(C5I)	
$FFE0,$FFE1	Enhanced Capture Timer channel 7	I-Bit	TIE(C7I)	$E0
$FFDE,$FFDF	Enhanced Capture Timer overflow	I-Bit	TSRC2(TOF)	$DE
$FFD2,$FFD3	ATD0	I-Bit	ATD0CTL2(ASCIE)	$D2
$FFD0,$FFD1	ATD1	I-Bit	ATD1CTL2(ASCIE)	$D0
$FFCE,$FFCF	Port J	I-Bit	PTJIF(PTJIE)	$CE
$FFCC,$FFCD	Port H	I-Bit	PTHIF(PTHIE)	$CC
$FF8E,$FF8F	Port P	I-Bit	PTPIF(PTPIE)	$8E

Interrupts werden durch *Unterbrechungswerk* (Interrupt Unit, *Interrupt Controller*) koordiniert.
Interupts sind meist *priorisiert*.
Interrupts können „*maskiert*“ werden, sie kommen dann nicht zum Zug.
Bei den meisten Rechnern besitzt der Zentralprozessor nur wenige Interrupteingänge:

- *vorgeschaltete Interrupt Controller* fächern Interrupt auf
- Prozessor muss mit Interrupt Controller kommunizieren, um Unterbrechungsquelle zu erfahren

- *Unterbrechungsquelle* ist fest einem *Interrupt-Vektor zugeordnet*

Im 68HCS12 können maskierbare Interrupts mit speziellem Assemblerbefehl kollektiv durch *Setzen des I-Bits* im Condition Code Register CCR blockiert (maskiert) werden (I-Bit=1).

Für jeden maskierbaren Interrupt kann an der Quelle über ein Bit bestimmt werden, ob der Interrupt aktiviert ist (Bit = 1) oder deaktiviert ist (Bit = 0).

Interrupts können von höher priorisierten Interrupts unterbrochen werden (*Mehrfachunterbrechung*):

3.9.1 Vorbereitung und Ablauf eines Interrupts (Beispiel 68HCS12)

Damit ein Interrupt verarbeitet werden kann, müssen einige Voraussetzungen erfüllt sein:

- Interruptquelle muss einen Interrupt anzeigen. Für die eingebauten Peripheriegeräte geschieht das durch das Setzen eines Interrupt-Flags in einem zugehörigen Statusregister.
- Interrupts müssen an der Quelle *freigeschaltet* sein. Für die eingebauten Peripheriegeräte geschieht das durch Setzen eines Enable-Bits in einem zugehörigen Steuerregister (Spalte „Enable“ in Tabelle 3-6).
- Interrupt darf nicht *maskiert* sein. In den meisten Fällen bedeutet dies, dass das I-Bit im CCR zurückgesetzt sein muss. Nach dem Einschalten des Rechners steht das I-Bit auf 1, d.h. alle maskierbaren Interrupts sind gesperrt. Das Monitorprogramm setzt das I-Bit aber auf 0, da es Interrupts für die serielle Schnittstelle benötigt.

Was passiert nun, wenn alle oben erwähnten Bedingungen erfüllt sind und ein Interrupt durchgeleitet worden ist? Der Prozessor geht wie folgt vor:

- Der Programmzähler PC, der auf die nächste auszuführende Anweisung zeigt, die Register Y,X,A,B und das CCR werden in genau dieser Reihefolge auf den Stack gelegt
- Das I-Bit im CCR wird gesetzt; somit sind alle weiteren maskierbaren Interrupts blockiert. Die Interruptroutine kann also nicht von einem weiteren maskierbaren Interrupt unterbrochen werden. Im Falle eines XIRQ-Interrupts wird auch noch das X-Bit im CCR gesetzt.
- Der Programmzähler wird mit den zugehörigen Interruptvektor geladen, d.h. der Rechner springt zu

der durch den Interruptvektor definierten Adresse und arbeitet den dort liegenden Befehl ab.

- Der Rechner arbeitet die weiteren Befehle der Interrupt-Serviceroutine ab.

Die letzte Anweisung in einer Interrupt-Serviceroutine ist immer die `RTI` (Return from Interrupt)-Anweisung. Diese Anweisung restauriert alle vor Eintritt in den Interrupt auf den Stack gelegten Register.

Häufig ist es nicht akzeptabel, dass ein niedrig priorisierter Interrupt die Bearbeitung eines wichtigeren Interrupts verhindert. Aus diesem Grund ist es üblich, die Maskierung durch das I-Bit so schnell wie möglich wieder aufzuheben. Dies geschieht entweder durch sehr kurze Interruptroutinen, oder indem man innerhalb der Interruptroutine den `CLI`-Befehl (Clear Interrupt) ausführt.

Bevor man den `CLI`-Befehl aufruft, muss sichergestellt sein, dass das Interruptflag zurückgesetzt ist, sonst würde sofort wieder der gleiche Interrupt auftreten.

3.9.2 Beispiel: Interrupt-Serviceroutine in C für Freescale-Rechner

Interrupt-Serviceroutinen lassen sich in Assembler oder in C programmieren. Wir nehmen unser Blinklicht-programm und schauen uns an, wie es in der Programmiersprache C aussieht (in der Materialsammlung unter Beispiele\Interrupt-C):

```
1   void main(void) {
2      EnableInterrupts;  /* Das brauchen wir für den Debugger und den RTI */
3      /* Deaktiviere die 7-Segment Anzeige */
4      DDRP = DDRP | 0xf; /* Data Direction Register Port P */
5      PTP  = PTP | 0x0f; /* Schalte alle vier Segmente aus */
6
7      /* Aktiviere die LEDs */
8      DDRJ_DDRJ1 = 1;   /* Data Direction Register Port J */
9      PTJ_PTJ1   = 0;  /* Schalte LED-Zeile ein
10
11     /* Schalte Port B als Ausgang */
12     DDRB        = 0xFF; /* Data Direction Register Port B */
13
14     /* Schalte LED PB0 ein und aus */
15     PORTB = 0x01;
16  /*...... */
```

Bis hierhin unterscheidet sich das Programm überhaupt nicht von unserem ersten einfachen Blinklichtprogramm mit Warteschleife. Jetzt jedoch müssen wir den Teiler für den Echtzeitzähler einstellen und den Echtzeit-Interrupt aktivieren:

```
17    /* Stelle Teiler für RTI ein */
18    RTICTL = 0x7f;
19    CRGINT = CRGINT | 0x80;  /* schalte RTI frei */
20
21    /* Und hier kommt das Hauptprogramm... */
22    for(;;) {
23      /* Däumchen drehen, Unterbrechungsroutine macht alles für uns ... */
24    }
25  }
```

Was übrig bleibt, ist die *Interrupt-Serviceroutine*.

In Programmiersprache C gibt es *kein standardisiertes Schlüsselwort*, um eine Routine als *Interruptroutine* zu *kennzeichnen*. Compiler muss dies jedoch wissen, da er sonst statt eines `RTI`-Befehls am Ende einen `RTS`- oder `RTC`-Befehl generieren würde.

Fast alle Compiler für eingebettete Systeme *kennzeichen Unterbrechungsroutinen* durch das *Schlüsselwort* „*interrupt*“ oder eine compilerspezifische pragma-Anweisung (im CodeWarrior-Compiler heißt sie z.B. `#pragma TRAP_PROC` und wird direkt vor die Routine gesetzt).

Zuordnung der Unterbrechungsroutine zu Unterbrechungsquelle:

- entweder in Linker-Befehlsdatei
- als Nummer hinter Schlüsselwort interrupt (Vektoren sind durchnummeriert)

Wir benutzen lieber „interrupt“ mit Nummer (keine Verteilung auf zwei Dateien) und schreiben:

```
26  interrupt 7 void rtiISR (void) {
27     CRGFLG = CRGFLG | 0x80;      /* setze Interrupt-Flag zurueck */
28     PORTB = ~PORTB & 0x01;       /* schalte LED ein/aus */
29  }
```

3.10 Software-Simulation und Test

Typisches Szenario: Software muss entwickelt werden, aber noch keine Hardware-Labormuster da.

Wichtig: automatisierte Regressionstests. Mit Hardware nicht immer realisierbar.

Forderung: Applikations-Software sollte auf Standard-Hardware (PC, Workstation) ablauffähig sein. Vorteile:

- Effiziente Entwicklungsumgebung.
- Sehr gute Testbarkeit (Code Coverage, Speicherlöcher, Automatisierbarkeit, reproduzierbare Ergebnisse).
- Software-Entwicklung kann vor Bereitstellung der Hardware erfolgen.

Grenzen dieses Ansatzes:

- Echtzeitverhalten nicht äquivalent zu Ablaufumgebung
- Verzahnung mit Betriebssystem nicht immer nachbildbar
- Intelligente Schnittstellen u.U. schwierig nachbildbar

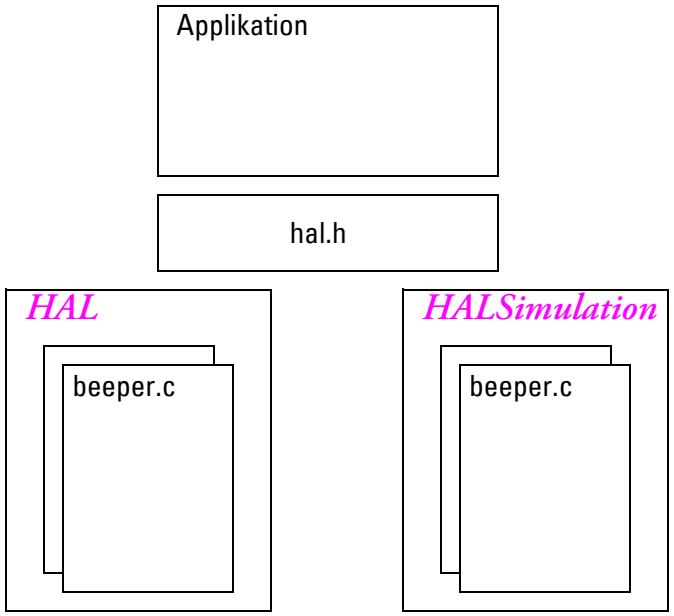
Trotzdem: wenn immer denkbar soll dieser Ansatz verwendet werden.

3.10.1 Strukturierung der Software

Vorgehensweise, wenn kein Betriebssystem involviert ist:

- Schnittstelle zu HAL definieren (z.B. `getTickCount()`, `getButtonPressed()`, `setLED()`, `beeperOn()`, `beeperOff()`, `writeLCD()`)
- Zwei Verzeichnisse: `HAL` und `HALSimulation`
- `HAL` enthält die echten Hardwareroutinen, `HALSimulation` die äquivalenten für die Simulation
- Aufrufschnittstelle identisch, d.h. es gibt nur ein `hal.h`
- Es gibt nur ein Exemplar der Quelldateien für die Anwendungssoftware auf dem Entwicklungsrechner (keine Kopien)
- Die Anwendungssoftware ist weitestgehend unabhängig davon, ob sie für das Zielsystem oder die Simulationsumgebung gebaut wird (keine `#ifdef` für diesen Zweck, außer wo unvermeidlich; nicht verstreut über den Quellcode, sondern an einer Stelle)

- Includierung von HAL oder HALSimulation wird über die Entwicklungsumgebung gesteuert.



3.10.2 Testen auf Host-System

Im realen System erzeugt das System selbst Ereignisse, z.B. Zeitgeber, oder Interrupts von den Tasten.
Das System erzeugt Ausgaben, z.B. auf das LCD oder den Beeper.
Das muss in der Simulation nachgebildet werden:

- Alle Ereignisse werden in eine Datei geschrieben.
- Jede Zeile setzt den Zeitgeber um einen oder mehrere Ticks vor.
- Beispiel für Tastendruck: in der Zeile steht eine „1“ für Taste 1
- Beispiel für 100 Ticks je 10 ms: „+100“

- Beispiel für einen Tick: „.“
- Ausgaben werden in eine Ausgabedatei geschrieben (z.B: „beep“, nicht PC-Lautsprecher tönen lassen, das ist nicht automatisch testbar)

Es darf mehrere Eingabedateien geben, um unterschiedliche Szenarien zu modellieren.

3.10.3 Portierung auf Zielsystem

Sind alle Tests erfolgreich gewesen, müssen die Funktionen für den realen HAL entwickelt werden.

- Funktionen müssen identische Schnittstelle wie die der Simulation haben.
- Funktionen müssen auf der Hardware getestet werden, sollten möglichst klein und unkompliziert sein.
- Danach können Sie mit Applikation zusammengebunden werden.

Applikation mit HAL kann komplett aufs Zielsystem gebracht und dort getestet werden.

3.11 Techniken für die Systemmodellierung

Wir unterscheiden Methoden zur Modellierung und zum Entwurf von Systemen in drei Klassen:

- Methoden zur Modellierung *zeitkontinuierlicher* Regelungen (Vorlesung Regelungstechnik)
- Methoden zur Modellierung *zeitdiskreter* Regelungen (Vorlesung Regelungstechnik)
- Methoden zur Modellierung und zum Entwurf *diskreter* Steuerungen (yeah)

Folgende Methoden zur Modellierung und zum Entwurf diskreter Steuerungen werden verwendet:

- Petri-Netze (eher im akademischen Bereich)
- Zustandsdiagramme, insbesondere Harel-Statecharts, UML-Zustandsdiagramme

Aus Zeitgründen konzentrieren wir uns auf die in der Praxis verbreiteten UML-Zustandsdiagramme.

3.12 UML-Zustandsdiagramme

UML-Zustandsdiagramme sind mit die nützlichste UML-Diagrammart für dedizierte Systeme. Aus UML-Zustandsdiagrammen lässt sich mit entsprechenden Werkzeugen ausführbarer Code erzeugen.

-

4

Echtzeit-Programmmierung

4.1 Problemstellung und Anforderungen

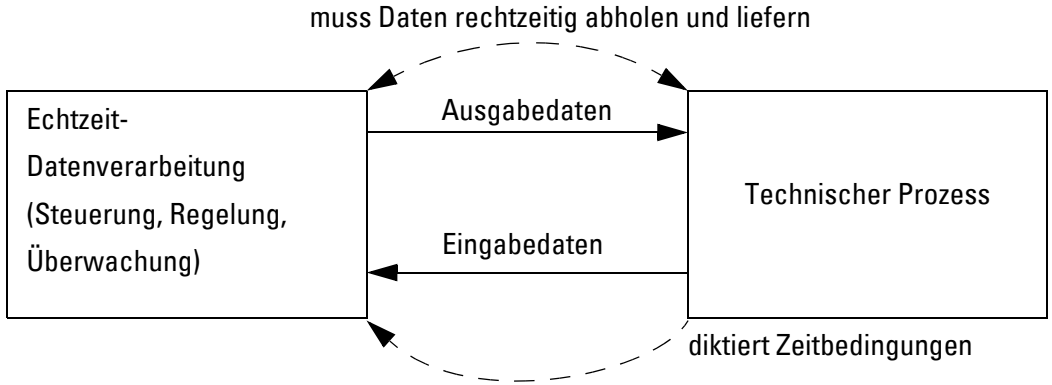
- Nicht-Echtzeitsysteme: *logische Korrektheit* bedeutet Korrektheit
- Echtzeitsysteme: *logische Korrektheit plus zeitliche Korrektheit* bedeutet Korrektheit

4.1.1 Rechtzeitigkeit

Bedeutung: Ausgabedaten müssen *rechtzeitig* zur Verfügung gestellt werden.

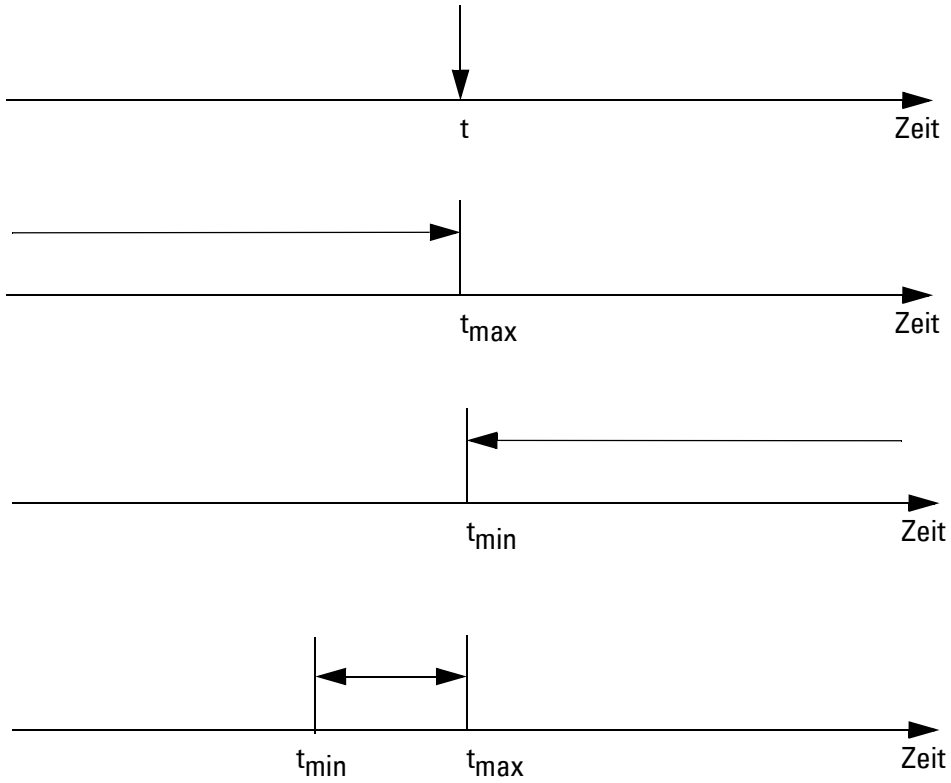
Erfordert indirekt auch die rechtzeitige Abholung von Eingangsdaten.

Technischer Prozess *diktiert Zeitbedingungen*.



Verschiedene Formen der Zeitbedingungen:

- Angabe eines ***genauen Zeitpunktes*** (Aktion muss genau zu diesem t stattfinden); ***Beispiel: Stoppen eines Fahrzeugs an einem genauen Punkt***
- Angabe eines ***spätesten Zeitpunktes*** (Zeitschranke, ***Deadline***)
- Angabe eines frühesten Zeitpunktes ***(z.B. Entladen nicht vor Beladen, wann spätestens ist egal)***
- Angabe eines Zeitintervalls (Aktion muss innerhalb dieses Zeitintervalls durchgeführt werden)



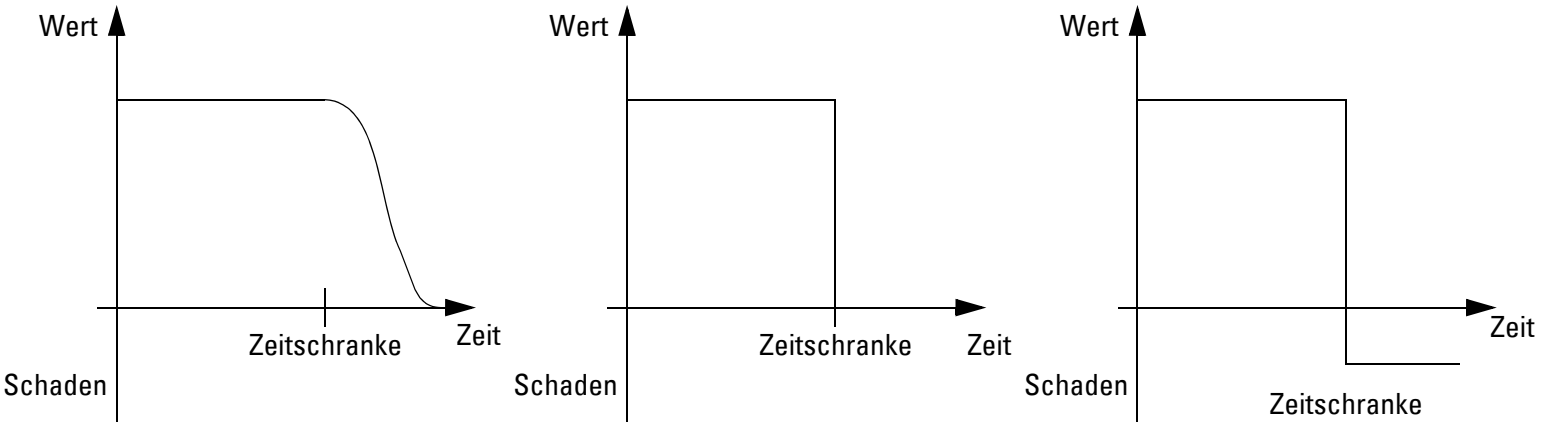
Unterteilung der *Zeitbedingungen* in

- *periodische* Zeitbedingungen
- *aperiodische* Zeitbedingungen

Weitere Unterteilung der *Zeitbedingungen* in

- *absolute* Zeitbedingungen
- *relative* Zeitbedingungen

Definitionen von harten, festen und weichen Echtzeitbedingungen:



weich *fest* *hart*

Drei verschiedene Ansätze für Kriterien:

- nach Kritikalität (entsteht Schaden oder nicht)*; Beispiel: Airbag, Videoprozessor*
- nach Nützlichkeit (wie nützlich sind zu späte Ergebnisse)*; Problem hier: die Nützlichkeitsfunktion*
- nach zulässiger Häufigkeit von Zuspätkommen*; z.B. muss in 99,999% aller Fälle Zeitschranke einhalten*

Zeitparameter:

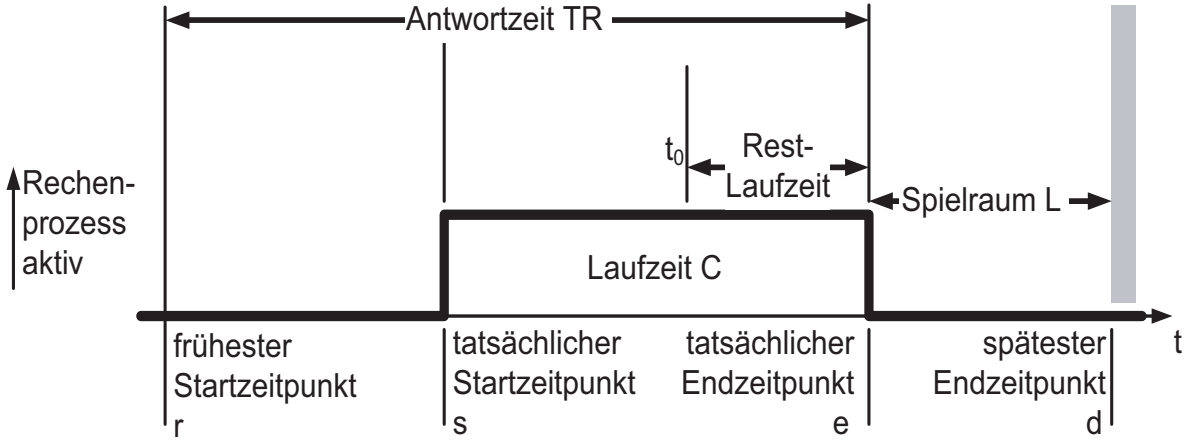
- *r*, release time, frühester Startzeitpunkt, Job wird ablaufbereit, hat z.B. alle Eingabewerte. Job kann irgendwann nach r zur Ausführung eingeplant werden. Job hat keine release time, wenn er bei t=0 ablaufbereit ist.
- *C*, worst case Ausführungs- oder Rechenzeit.
- *d*, Deadline (Zeitschranke, Frist), zu diesem Zeitpunkt muss ein Job fertig sein. Job hat also (deadline – release time) Zeit für Ausführung. Ein Job hat keine Zeitschranke, wenn die Zeitschranke bei unendlich liegt.
- Die Antwortzeit ist die Zeitspanne zwischen Release-Zeit und dem Zeitpunkt, wenn der Job fertig ist.
- *D*, die relative Zeitschranke, d.h. der maximal zulässige Wert für die Bereitstellung des Ergebnisses.
- *T*, Task-Periode (nur für periodische Tasks).
- Die absolute Deadline ist die Release-Zeit plus der relativen Deadline.
- Zeitbedingungen werden oft in Form von Einschränkungen der Release-Zeit und der relativen oder absoluten Deadline gesetzt.

Abgeleitete Zeitparameter:

- u = C/T ist der *Prozessorauslastung* einer Task.
- ch = C/D ist der *Prozessor-Lastfaktor* einer Task.
- s ist der Startzeitpunkt der Bearbeitung eines Jobs.
- e ist der Fertigstellungszeitpunkt eines Jobs.
- L = C-D ist der nominale Spielraum (laxity) einer Task.
- D(t) = d-t is the residual relative deadline at time t.
- C(t) ist die noch verbleibende Rechenzeit zum Zeitpunkt t.
- L(t) = C(t) – D(t) ist der verbleibende nominale Spielraum einer Task zum Zeitpunkt t

also: $L(t) = D + r - t - C(t)$

- TR = e-r ist die *Task-Antwortzeit*.



4.1.2 Gleichzeitigkeit

4.1.3 Verfügbarkeit

4.2 Verfahren

4.2.1 Synchrone Programmierung

4.2.2 Asynchrone Programmierung

4.3 Prozesse, Threads und Tasks

4.4 Ablaufsteuerung

4.4.1 Zyklische Ablaufsteuerung

4.4.2 Zeitgesteuerte Ablaufsteuerung

4.4.3 Unterbrechungsgesteuerte Ablaufsteuerung

4.4.4 Einplanung und Einlastung

Sobald mehrere Prozesse auf einer CPU ablaufen sollen, muss entschieden werden, wann welcher Prozess wieviel Rechenzeit erhält. Diesen Vorgang unterteilen wir konzeptionell in zwei Schritte: die ***Einplanung*** (engl. *scheduling*) und die ***Einlastung*** (engl. *dispatching*). Die Einplanung stellt die Strategie dar, nach der die einzelnen Arbeitsaufträge in eine zeitliche Reihenfolge eingelastet werden sollen.

Wir unterteilen Einplanungsverfahren in zwei Typen:

- bei der ***gekoppelten Einplanung*** findet die Planung zur Laufzeit des System statt (*on-line scheduling*). Diese Art der Einplanung nennt man auch ***dynamische Einplanung***.
- bei der ***entkoppelten Einplanung*** findet die Planung außerhalb des Systems oder vor dem Systemstart statt (*off-line scheduling*), z.B. schon beim Systementwurf oder auf einem anderen Rechner. Diese Art der Einplanung nennt man auch ***statische Einplanung***.

Die Einlastung ist der Mechanismus, nach dem der durch die Einplanung erstellte Plan umgesetzt wird. Die Einlastung ist immer mit der Aufgabenabarbeitung gekoppelt. In der Praxis wird bei vielen Echtzeitbetriebssystemen nicht zwischen Einplanung und Einlastung unterschieden.

Gekoppelte Einplanung (online-scheduling) erlaubt es, flexibel auf an das System herangetragene Arbeitsaufträge zu reagieren. Es ist nicht erforderlich, schon beim Systementwurf zu wissen, welche Ereignisse wann auf das System treffen. Diese Strategie erfordert allerdings, entsprechende Rechenzeitreserven vorzuhalten und erlaubt eine nur eingeschränkte Auslastung des Systems. Aufgrund der Flexibilität ist sie jedoch die in der Praxis dominante Einplanungsstrategie.

Entkoppelte Einplanung (offline-scheduling) bietet sich vor allem dort an, wo die abzuarbeitenden Arbeitsaufträge einschließlich der durch sie entstehenden Rechenlast vor dem Systemstart bekannt sind. Das System muss in einem solchen Fall deterministisch sein. Ändern sich die der Planung zugrunde liegenden Annahmen, muss ein neuer Ablaufplan entworfen und in das System eingespielt werden. Entkoppelte Einplanung verwendet man deshalb vorwiegend in sicherheitskritischen Bereichen.

Die statische Einplanungsstrategie ist oft als taktgesteuerte (*clock driven*) oder zeitgesteuerte (*time driven*) Einplanung realisiert. In diesem Fall findet die Einlastung zu fest vorgegebenen Zeitpunkten statt.

Die dynamische Einplanungsstrategie ist oft als vorranggesteuerte (*priority driven* oder *event driven*) Einplanung realisiert. Dabei wird der Ablaufplan mit jedem Eintreffen eines Ereignisses neu erstellt.

Darüber hinaus findet man in der Praxis sogenannte zyklische Einplanungsstrategien, die als Zeitscheibenverfahren (*round-robin*) oder einfache zyklische Ablaufsteuerungen (*cyclic executive*) realisiert sind.

Ist der Dispatcher in der Lage, einen laufenden Arbeitsauftrag zurückzustellen und durch einen anderen Arbeitsauftrag zu verdrängen, spricht man von einem präemptiven System. Präemptivität kann als Attribut von Arbeitsaufträgen implementiert sein, d.h. im gleichen System kann es verdrängbare und nicht verdrängbare Aufträge geben (z.B. im OSEK-Betriebssystem). Tasks können an beliebigen Stellen unterbrochen werden (*fully preemptive*), an extra dazu ausgewiesenen Stellen (***preemption points***) oder gar nicht (*run to completion*).

Unterstützt ein Betriebssystem präemptive Tasks, spricht man von einem voll ***präemptiven Betriebssystem***. Wenn die Tasks selbst die Kontrolle an das Betriebssystem zurückgeben müssen, um dem Scheduler und Dispatcher die Möglichkeit einzuräumen, tätig zu werden, spricht man von einem ***kooperativen Betriebssystem***.

4.4.5 Einfache zyklische Verfahren

4.4.5.1 Zyklische Ablaufsteuerung mit Endlosschleife

4.4.5.2 Basic Cyclic Executive („Round Robin“)

Das Round-Robin-Verfahren entspricht im Wesentlichen einer Endlosschleife, bei der den existierenden Tasks der Reihe nach zyklisch Rechenzeit zugeteilt wird.

Sind die einzelnen Tasks präemptiv, kann die Einlastung in vorgegebenen Zeitschlitzen erfolgen, d.h. reihum erhält jede Task einen Zeitschlitz um abzulaufen. Wird sie innerhalb dieses Zeitschlitzes nicht fertig, wird sie unterbrochen, und die nächste ablaufbereite Task wird eingelastet. Die Arbeit an der unterbrochenen Task wird erst wieder aufgenommen, wenn alle anderen ablaufbereiten Tasks einen Zeitschlitz nutzen konnten. Die Länge des Zeitschlitzes nennt man auch „Quantum“. Das Quantum kann man in der Regel konfigurieren. Wird es zu kurz gewählt, muss das System viele Taskwechsel durchführen und reduziert durch die damit

verbundenen Verwaltungskosten die effiziente Nutzung der Recheneinheit. Wird das Quantum zu groß gewählt, ist die Reaktionszeit auf interaktive Eingaben zu langsam. Ein Quantum zwischen 5 und 10 ms ist bei einem Echtzeitsystem in der Regel ein guter Kompromiss.

Ist eine Task nicht präemptiv, kann erst zur nächsten Task weitergeschaltet werden, wenn die laufende Task an ihr Ende gekommen ist bzw. die Kontrolle freiwillig an das Betriebssystem zurückgegeben hat.

Das Round-Robin-Verfahren mit präemptivem Multitasking sorgt für eine faire Zuteilung von Rechenzeit. Es kann nicht berücksichtigen, dass bestimmte Tasks eine höhere Dringlichkeit haben als andere. In manchen RTOS ist es möglich, den Tasks eine unterschiedliche Anzahl an Zeitschlitzen pro Zyklus zuzugestehen, um z.B. Tasks mit längeren Verarbeitungszeiten adäquat zu berücksichtigen.

4.4.6 Takt- und zeitgesteuerte Verfahren

4.4.6.1 Zeitgesteuerte Ablaufsteuerung

Für manche Anwendungen ist die „Echtzeit“, die das Round-Robin-Verfahren bietet, nicht genau genug. Das Round-Robin-Verfahren bemüht sich, jede Task so schnell und oft wie möglich laufen zu lassen. In komplexeren Applikationen ist es aber oft wichtiger, Termine genau einzuhalten, anstatt möglichst schnell zu

sein. Zum Beispiel gibt es Anwendungen in der digitalen Signalverarbeitung und Regelungstechnik, die Daten mit einer bestimmten Taktrate verarbeiten müssen. Die unregelmäßigen Periodendauern der Endlosschleife oder des Round-Robin-Verfahrens würden zu eine Fehlfunktion des Systems führen.

Ein Time Driven Cyclic Executive Scheduler stößt initiiert vom Interrupt eines Hardware-Timers zyklisch eine Taskfolge an. Es muss sichergestellt sein, dass diese Taskfolge vollständig abgearbeitet werden kann, bevor der nächste Interrupt vom Hardware-Timer kommt. Der Hardware-Timer legt die Rate fest, mit der die Taskfolge ausgeführt wird.

4.4.7 Vorranggesteuerte Verfahren

4.4.7.1 Unterbrechungsgesteuerte Ablaufsteuerung

4.4.7.2 Zyklische Ablaufsteuerung mit Unterbrechungen

4.4.7.3 Verfahren mit statischer Prioritätsvergabe

Prioritätsbasierte präemptive Zeitplanverfahren sind die meistgenutzten Verfahren in Echtzeitbetriebssystemen. Dieses Konzept stellt sicher, dass immer, wenn das System ein Ereignis bemerkt, der Scheduler die ablaufbereite Task mit der höchsten Priorität einplant und der Dispatcher sie einlastet. Erst wenn diese Task blockiert oder sonstwie ihre Aktivität vollständig beendet hat, wird die Task mit der nächstniedrigeren Priorität eingelastet.

Wenn es zulässig ist, dass Tasks die gleiche Priorität erhalten dürfen, konkurrieren diese um die Rechenzeit. Den daraus entstehenden möglichen Konflikt löst man häufig mit einem präemptiven oder kooperierenden Round-Robin-Verfahren. Tasks gleich hoher Priorität werden der Reihe nach eingelastet, entweder innerhalb bestimmter Zeitscheiben oder bis sie blockieren oder sonstwie ihre Aktivität beendet haben.

Zur Implementierung eines prioritätsbasierten, präemptiven Zeitplanverfahrens bedarf es zweier Elemente: eines Betriebssystemkerns, der Präemption unterstützt, sowie eines Schedulers, der immer die ablaufbereite Task mit der höchsten Priorität dem Dispatcher meldet.

\begin{figure}[htbp]

```
  \begin{center}
   \includegraphics[scale=0.7]{scheduling/roundRobinWithPriority.eps}
    \caption{\label{fig:roundRobinWithPriority}
          Prioritätsbasiertes Zeitplanverfahren mit Round-Robin}
  \end{center}
\end{figure}
```

Da ein prioritätsbasierter Scheduler versucht, immer der ablaufbereiten Task mit der höchsten Priorität die Recheneinheit zu überlassen, muss er über alle Tasks im System mindestens zwei Dinge wissen: den momentanen Zustand (laufend, ruhend, ablaufbereit, wartend) und die Priorität.

Um dann die Einlastung veranlassen zu können, muss von jeder Task auch noch bekannt sein, wo ihr Kontext, d.h. wo der zuletzt aktuelle Registersatz liegt. Diese Informationen nennt man den **Task Control Block** oder **TCB**. Der Scheduler verwaltet alle TCBs der einzelnen Tasks in einer Liste, der **Ablaufwarteschlange.**

```
\begin{figure}[htbp]
  \begin{center}
   \includegraphics[scale=0.7]{scheduling/tcb.eps}
    \caption{\label{fig:tcb}
```

```
        Struktur eines Task Control Blocks}
  \end{center}
\end{figure}
```

In RMOS ist die Task-Liste eine statische Liste der TCBs aller Tasks, geordnet nach Task-ID. Damit kann auf eine Task über deren Task-ID zugegriffen werden.

```
\begin{figure}[htbp]
  \begin{center}
   \includegraphics[scale=0.7]{scheduling/taskliste.eps}
    \caption{\label{fig:taskliste}
        Lineare Taskliste in RMOS}
  \end{center}
\end{figure}

\begin{figure}[htbp]
```

```
  \begin{center}
   \includegraphics[scale=0.7]{scheduling/ablaufwarteschlange.eps}
    \caption{\label{fig:ablaufwarteschlange}
          Beispiel einer Ablaufwarteschlange}
  \end{center}
\end{figure}
```

Die Ablaufwarteschlange wird automatisch immer dann vom Betriebssystem aktualisiert, wenn ein Aufruf einer Betriebssystemfunktion zu einer Änderung der Priorität oder des Zustands einer Task führt, also z.B. wenn eine Task auf die Freigabe einer Semaphore oder ein Event Flag wartet.

```
\begin{figure}[htbp]
  \begin{center}
   \includegraphics[scale=0.7]{scheduling/abwsAlternativ.eps}
    \caption{\label{fig:abwsAlternativ}
```

```
        Ablaufwarteschlange, alternative Sicht}
  \end{center}
\end{figure}
```

Die Rolle der Ablaufwarteschlange bei einem Taskwechsel sei an einem Beispiel illustriert. Nach Abbildungen \ref{fig:ablaufwarteschlange} und \ref{fig:abwsAlternativ} werde Task T2 beendet, d.h. in den Zustand ruhend gebracht. Dies führt zu einem Taskwechsel nach T4. Dabei werden folgende Schritte durchlaufen:

```
\begin{enumerate}
  \item EXIT: T2 aus Ablaufwarteschlange ausfügen, Kontext (= Registersatz)
      auf Stack(T2) sichern, Stackpointer $\to$ TCB(T2), S(T2) := (R)uhend

  \item Suche nach der höchstprioren Task in der Ablaufwarteschlange: T4

  \item Zuteilung der CPU an die neue laufende Task T4
      S(T4) := (L)aufend, Stackpointer aus TCB(T4) laden, Kontext von
      Stack(T4) in Register laden
\end{enumerate}
```

Die Bearbeitung der Ablaufwarteschlange durch das Betriebssystem ist Rechenzeit, die für die Anwendung verloren geht. Idealerweise sollten diese Vorgänge möglichst keine Rechenzeit benötigen. Es ist deshalb eine besondere Herausforderung für den Entwickler eines Betriebssystems, das Ausfügen (1) und die Suche (2) nach dem nächsten höchstpriorisierten ablaufbereiten Task möglichst schnell zu machen. In der Vorlesung schauen wir uns an, wie diese Aufgabe für das Betriebssystem μC/OS-II gelöst wurde.

```
\begin{lstlisting}[caption=Der Scheduler von $\mu$C/OS-II]
```

Rate Monotic Verfahren

Wird ein Echtzeitzeitsystem mit prioritätsbasiertem präemptivem Zeitplanverfahren entworfen, stellt sich die Frage, welchen Tasks welche Priorität zuzuordnen ist und ob sichergestellt ist, dass die Echtzeitbedingungen immer eingehalten werden. Diese Fragestellung wurde schon 1973 von Liu und Layland für periodisch abzuarbeitende Tasks untersucht. Sie definierten die einfache Regel des „Rate-Monotonic Scheduling“ (RMS):

Tasks mit kleinerer Periode erhalten höhere Priorität.

Sie konnten zeigen, dass wenn es einen Plan mit festen Prioritäten gibt, der alle Termine einhält, sich dieser mit RMS erzeugen lässt.

Es gibt Taskmengen, für die keine Lösung existiert, weil sie zu einer CPU-Last von mehr als 100 % führen würden. Wir definieren die CPU-Auslastung als

$$A = \sum^{N}_{i=1} TV_i/TP_i$$

Es kann keine Lösung für die Rechenzeitzuteilung geben, wenn $A > 1$. Das Ergebnis der Untersuchung zu RMS ergab folgende Bedingung für die Lösungsmöglichkeit, n Tasks Rechenzeit zuzuteilen und dabei die Echtzeitbedingungen einzuhalten:

$$A < n (2^{1/n}-1)$$

```
\begin{table}[h]
 \centering
 \begin{tabular}{lll} \toprule
  n      & Grenze für $A$ &\\ \midrule
  1      & 100\%          & Trivial: nur eine Task\\
  2      &  83\%          & Zwei Tasks \\
  3      &  78\%          & \\
  4      &  76\%          & \\
  $\infty$    &  69\%          & asymptotischer Wert\\
```

```
   \bottomrule
  \end{tabular}
  \caption{Planbarkeit bei $n$ periodischen Tasks}
  \label{tab:RMSBound}
\end{table}
```

Dies bedeutet, dass Echtzeitbedingungen bei periodischen Tasks immer eingehalten werden, wenn die Prozessorauslastung A kleiner als 69 % ist. Umgekehrt gilt nicht, dass es keine Lösung gibt, wenn die Prozessorauslastung höher liegt als 69 %: es kann für diesen Fall trotzdem noch eine Lösung geben, aber das ist nicht mehr garantiert.

Das RMS Zeitplanverfahren ist stabil insofern, dass selbst wenn immer mehr niedriger priorisierte Tasks zum System hinzugefügt werden, die höher priorisierten Tasks ihre Fristen weiterhin einhalten, auch wenn die niedriger priorisierten Tasks nicht mehr fristgerecht arbeiten.

Die Gleichung \ref{eq:cpuauslastung3} kann ergänzt werden, falls niedriger priorisierte Tasks höher priorisierte für eine bestimmte maximale Zeit B_j blockieren können in

```
\sum^{N}_{i=1} TV_i/TP_i + max(\frac{B_1}{TP_1},\ldots,
\frac{B_n}{T_n})
< n (2^{1/n}-1)
```

Das RMS-Verfahren kann eingesetzt werden, wenn folgende Voraussetzungen gegeben sind:

1. Jeder periodische Prozess muss in seiner Periode fertig werden.
2. Kein Prozess hängt von einem anderen ab.
3. Jeder Prozess benötigt den gleichen Anteil an CPU-Zeit in jedem Schlitz.
4. Kein nicht-periodischer Prozess hat eine Deadline.
5. Die Unterbrechung von Prozessen tritt sofort und ohne zusätzlichen Aufwand ein.

Auch wenn die letzte Bedingung nicht sehr realistisch ist, vereinfacht sie die Modellierung des Systems wesentlich ohne großen Genauigkeitsverlust. Hat man z.B. ein System mit N Prozessen, von denen jeder mit einer bestimmten Frequenz f_x auftritt, kann man nach dem RMS-Verfahren die Prioriäten proportional den Frequenzen festlegen.

4.4.7.4 Deadline Monotonic Verfahren

4.4.7.5 Verfahren mit dynamischer Prioritätsvergabe

Earliest Deadline First Verfahren

Die Zuteilung der Prioritäten auf die Tasks erfolgt nach dem RMS-Verfahren statisch, d.h. die Prioritäten ändern sich zur Laufzeit nicht. Hat man drei Hausaufgaben zu bearbeiten, von denen eine nicht sehr wichtig ist, aber in einer Stunde abgegeben werden muss, wird man zunächst an diese gehen. Wenn alle zum gleichen Zeitpunkt abzugeben sind, wird man die wichtigste zunächst bearbeiten.

Dies illustriert ein Problem beim prioritätsbasierten Scheduling: es wird nicht berücksichtigt, wann etwas fällig ist sondern nur, wie wichtig es insgesamt ist. Es müssten also Tasks, die in Kürze fällig sind, vorgezogen werden vor höherprioren Tasks, die aber noch lange Zeit bis zum Ablauf ihrer Frist haben.

Der Algorithmus des Planens nach frühester Frist (engl. *Earliest Deadline First* (EDF)) weist dynamisch die höchste Priorität dem Task zu, dem am wenigsten Zeit bis zum Ablauf seiner Frist bleibt.

Dieses Verfahren garantiert, dass wenn es überhaupt eine Lösung basierend auf dynamisch vergebenen Prioritäten gibt, diese erzielt wird. Eine Lösung basierend auf EDF existiert, wenn die Prozessorauslastung A kleiner als 100 % ist.

```
\begin{figure}[htbp]
 \begin{center}
  \includegraphics[scale=1.0]{scheduling/7-12.eps}
   \caption{\label{fig:rmsundedf}
```

```
        Beispiel für RMS und EDF-Scheduling}
 \end{center}
\end{figure}
```

Obwohl dieser Algorithmus eine bessere CPU-Auslastung erlaubt, hat er in der Prozessdatenverarbeitung und anderen eingebetteten Systemen keine große Bedeutung erlangt. Dafür gibt es zwei wesentliche Ursachen. Der erste Grund liegt in der Instabilität des Algorithmus.

Wenn die Prozessorauslastung über 100 % steigt, wird mindestens ein Task ihre Termine nicht mehr einhalten können. Man kann aber nicht mehr sagen, welche das sein wird. Der zweite Grund für die geringe Verbreitung liegt in dem hohen Verwaltungsaufwand, der bei jedem Taskzustandswechsel betrieben werden muss, um die nächste ablaufbereite Task basierend auf ihrer Frist zu finden.

```
\begin{figure}[htbp]
 \begin{center}
  \includegraphics[scale=1.0]{scheduling/7-13.eps}
   \caption{\label{fig:rmsundedf2}
        Weiteres Beispiel für RMS und EDF-Scheduling}
 \end{center}
\end{figure}
```

Abbildung \ref{fig:rmsundedf} zeigt ein Beispiel für RMS- und EDF-Scheduling. Das System muss drei Prozesse A, B und C bearbeiten. Prozess A läuft alle 30 ms, z.B. ein MPEG-Videostream. Jeder Rahmen benötigt 10 ms Rechenzeit. Der Termin ist dadurch gegeben, dass die Bearbeitung des Rahmens beendet sein muss, bevor der nächste Rahmen bearbeitet werden kann.

Die Prozesse B und C laufen 25 Mal pro Sekunde (z.B. ein anderer MPEG-Videostream mit anderer Datenrate), bzw. 20 Mal pro Sekunde. Prozess B benötigt 15 ms Rechenzeit, Prozess C 5 ms Rechenzeit.

Abbildung \ref{fig:rmsundedf2} zeigt das gleiche Szenario wie zuvor beschrieben, mit dem einzigen Unterschied, dass Prozess A statt 10 ms nun 15 ms Rechenzeit pro Periode benötigt. Theoretisch ist die CPU noch nicht überbelegt, allerdings gibt es beim RMS-Algorithmus nun Fristverletzungen. Der EDF-Algorithmus schafft es, alle Aufgaben ohne Fristverletzungen einzuplanen.

Latest Release Time First Verfahren

Least Slack Time First Verfahren

Der Spielraum für eine Task ist definiert als die Frist abzüglich der verbleibenden Rechenzeit (s. Abb. \ref{fig:periodischZeitdiagramm}). Beim Planen nach geringstem Spielraum sortiert der Scheduler die Tasks nach ihrem Spielraum, und gibt dem Task die höchste Priorität, der den geringsten Spielraum besitzt.

Dieser Algorithmus verlangt, dass neben der Kenntnis der Frist eines jeden Tasks wie beim Planen nach frühester Frist zusätzlich die verbleibende Restlaufzeit bekannt sein muss. Aus diesem Grund und wegen seiner Instabilität (es kann nicht im Voraus gesagt werden, welcher Task bei Überlast verdrängt wird) hat auch dieser Algorithmus keine große praktische Bedeutung erlangt.

5

Echtzeit-Betriebssysteme

5.1 Taskverwaltung

5.2 Ressourcenverwaltung

5.3 Ereignisbehandlung

5.4 Zeitgeber-Verwaltung

5.5 Interprozesskommunikation

5.6 Speicherverwaltung

Jedes Programm nutzt in irgendeiner Weise wahlfreien Zugriffsspeicher (RAM). Es gibt verschiedene Möglichkeiten, diesen Speicher zu verwalten und zuzuteilen. Die wesentlichen Mechanismen sind

- Statische Allokation des Speichers
- Stack-basierte Verwaltung des Speichers
- Heap-basierte Verwaltung des Speichers

Diese Mechanismen schließen sich nicht gegenseitig aus, sondern werden meist in Kombination verwendet. Sie sollen im folgenden detaillierter besprochen werden.

5.6.1 Statische Speicherallokation

Wenn der zur Laufzeit benötigte Speicher schon bei der Erstellung eines Programms bekannt ist und zugewiesen wird, spricht man von statischer Speicherallokation. Diese Form der Speicherzuteilung hat den Vorteil, dass es zur Laufzeit keine Überraschungen aufgrund fehlender Speicherressourcen geben kann. Sie kann vor allem dort eingesetzt werden, wo der Rechner keine ausreichende Hardwareunterstützung für eine stackbasierte Speicherverwaltung zur Verfügung stellt. Der ursprüngliche 8051-Mikroprozessor hat zum Beispiel nur einen 8 Bit großen Stackzeiger, der lediglich die 256 Byte internes RAM adressieren kann. Da das interne RAM auch noch für andere Zwecke verwendet werden muss, reicht der Platz auf dem Stack oft nicht aus, um dort lokale Daten unterzubringen.

In Programmen sind alle Daten entweder global, statisch oder lokal innerhalb einer Funktion. Für globale und statische Daten muss beim Start des Rechners Platz im Speicher zugeteilt werden, der sich für den Rest der Laufzeit nicht mehr ändert. Diese Zuordnung ist unabhängig von der Art der Speicherverwaltung. Die Unterschiede zeigen sich im Umgang mit lokalen bzw. dynamisch zur Laufzeit entstehender Daten.

Das Verfahren der statischen Speicherallokation weist allen lokalen Daten einen Speicherblock zu, der für jede Funktion reserviert wird. Hat eine Funktion eine lokale Variable `v`, dann ist `v` bei jedem Funktionsaufruf immer an der gleichen Stelle im Speicher untergebracht. Arbeitet der Rechner diese Funktion nicht ab, ist die Speicherstelle ungenutzt.

Dieser Ansatz schließt die Verwendung von Rekursionen aus, da für jede Funktion nur einmal Speicherplatz für ihre lokalen Variablen reserviert wird.

Es gibt optimierende Compiler und Linker, die feststellen können, dass Funktionen nicht gleichzeitig aktiv sein können. Das bedeutet, dass solche Funktionen nie in der gleichen Aufrufhierarchie liegen dürfen (siehe Illustration in Bild 5-1).

Bild 5-1 Funktionen in Aufrufhierarchie

Um diese Analyse zur Kompilier- bzw. Linkzeit durchführen zu können, müssen zwei Einschränkungen in Kauf genommen werden. Erstens dürfen Funktionen nicht über ihre Funktionszeiger dynamisch aufgerufen werden, und zweitens dürfen solche Funktionen nicht von Unterbrechungsroutinen aus aufgerufen werden.

Werden diese Bedingungen eingehalten, dann dürfen sich die statisch allokierten Speicherbereiche überlappen, und der Speicherplatz kann besser genutzt werden.

Für größere Systeme ist das Verfahren der statischen Allokation nicht effizient, da die Menge an benötigtem Speicher mit jeder neuen Funktion wächst und der zur Verfügung stehende Speicher nur schlecht ausgenutzt wird. In der Praxis bieten moderne 8-Bit-Mikroprozessoren eine ausreichende Unterstützung für eine stackbasierte Speicherverwaltung, so dass die statische Allokation an Bedeutung verliert.

5.6.2 Stack-basierte Speicherverwaltung

Stellt die Hardware einen Stackpointer mit entsprechenden Befehlen zu seiner Manipulation (wie `push` und `pop`) sowie zur Stackpointer-relativen Adressierung zur Verfügung, können lokale Variablen auf einem Stapelspeicher (Stack) untergebracht werden. Beim Abspeichern eines Datums auf dem Stack mit einer `push`-Operation wird der Stackpointer automatisch von der Hardware auf die nächste freie Adresse des Stacks

geführt. Beim Zurückholen eines Datums vom Stack mit einer `pop`-Operation wird von der Hardware automatisch der Stackpointer zurückgeführt und die belegte Speicheradresse freigegeben.

Je nach Typ des Mikroprozessors kann der Stapelspeicher in Richtung höherer Adressen wachsen, oder in Richtung niedrigerer. Die Anzahl Bytes, die mit einer `push`- oder `pop`-Operation zwischen CPU und Speicher bewegt werden, hängt ebenfalls vom Typ des Mikroprozessors ab und entspricht meistens der Wortbreite der internen Prozessorregister. Einige Mikroprozessoren unterstützen verschiedene Betriebsarten wie z.B. einen User-Modus und einen Kernel-Modus, und stellen für die unterschiedlichen Modi eigene Stackpointer zur Verfügung.

Es ist in den meisten Fällen nicht möglich, zur Kompilierzeit die maximal erforderliche Stackgröße zu bestimmen. In einem Multitasking-System benötigt man für jede Task einen Stack, und zusätzlich eventuell einen weiteren für das Betriebssystem und die Unterbrechungsroutinen. Es muss sichergestellt sein, dass jeder Stack in allen Fällen ausreichend groß ist. Die kleineren 4-Bit, 8-Bit und 16-Bit Mikroprozessoren eingebetteter Systeme stellen in der Regel keine Hardwareunterstützung für eine virtuelle Speicherverwaltung zur Verfügung, von denen ein Stack im Bedarfsfall weiteren Speicher beziehen könnte.

In der Praxis dimensioniert man deshalb den Stack im fertigen System ca. 50% größer, als er im schlechtesten Fall während des Testens genutzt wurde. Dazu beobachtet man, wie weit der Stack im Test gewachsen war.

Eine einfache Technik dazu ist die, den Stackspeicher mit einem vorgegebenen Muster zu füllen, z.B. 0x55AA. Während der Stack wächst und wieder schrumpft, überschreibt er dieses Muster mit anderen Daten. Am Ende eines Tests kann man mit einer kleinen Schleife durch die Stackbereiche iterieren und so die Ausdehnung des Stacks feststellen.

Viele Echtzeitbetriebssysteme unterstützen einen solchen Mechanismus schon von Haus aus. Wenn nicht, oder wenn kein Echtzeitbetriebssystem eingesetzt wird, ist diese Technik auch leicht selbst zu implementieren. Sie kann sowohl während des Testens eingesetzt werden als auch in einem Produktionssystem, um den Softwareentwicklern frühzeitig Hinweise auf eine mögliche Unterdimensionierung eines Stacks zu geben. Die Messung auf einen Stacküberlauf muss nicht mit jeder Stackoperation durchgeführt werden; es genügt, den Test bei einem Warmstart, der z.B. nach einem Fehler passiert, ablaufen zu lassen.

Bild 5-2 Ermittlung der benötigten Stackgröße zur Laufzeit

Stacküberläufe können zu schwer auffindbaren Fehlern führen, wenn ein Stack in einen Speicherbereich hineinwächst, in dem dynamisch allokierte Daten abgelegt sind. Stacküberläufe können auch zu einem Systemabsturz führen, wenn dynamisch allokierte Daten den übergelaufenen Stack überschreiben und damit dort untergebrachte Rücksprungadressen zerstören.

Um die stackbasierte Speicherverwaltung muss sich der Programmierer ansonsten nicht weiter kümmern, der Compiler, die Rechnerhardware und das Betriebssystem verwalten diesen Speicher ohne weitere Information seitens des Programmierers. Vom Hochspracheprogramm aus sieht der Programmierer den Stack noch nicht einmal.

5.6.3 Heap-basierte Speicherverwaltung

Viele Objekte, Strukturen und Puffer erfordern eine Gültigkeitsbereich, der sich über mehr als eine einzelne Funktion oder ein einzelnes Modul erstreckt, der aber trotzdem nicht global sein soll. Zusätzlich ist die Größe des benötigten Speicherplatzes manchmal erst zur Laufzeit des Programms bekannt.

In der Programmiersprache C bieten die Funktionen `malloc()` und `free()` die Möglichkeit, solche dynamischen Speicherbereiche zu verwalten. Die Funktion `malloc()` gibt, wenn es möglich ist, dem Programmierer einen Zeiger auf einen Speicherbereich der gewünschten Größe zurück. Die Funktion `free()` gibt

den vorher allokierten Speicher wieder an das System zurück. So kann derselbe Speicherbereich während der Lebensdauer des Programms für unterschiedliche Zwecke verwendet werden. Der Programmierer muss sich während der Programmentwicklung keine Gedanken machen, welche Daten parallel benutzt werden, sondern kann über diese einfache Schnittstelle Speicher anfordern, wenn er benötigt wird.

Der für diese Funktion zur Verfügung stehende Speicher wird in einem Speicherbereich verwaltet, den man Heap nennt. Der Heap erstreckt sich in der Regel über den verbleibenden Speicher, der nicht vom Programmcode, statischen Daten und Stack belegt wird.

Der zur Verfügung stehende Speicher muss verwaltet werden. Die Speicherverwaltung soll selbst wenig Ressourcen benötigen, schnell und deterministisch arbeiten, und den zur Verfügung stehenden Speicher optimal ausnutzen.

In vielen dedizierten Systemen genügt die Standard-Implementierung von `malloc()` und `free()` nicht den Ansprüchen. Deshalb bieten die meisten Echtzeitbetriebssysteme eigene Routinen an, die prinzipiell die gleiche Funktion erfüllen, jedoch die im Echtzeitbereich und bei kleinen Systemen zusätzlichen Anforderungen berücksichtigen. Im folgenden werden exemplarisch zwei Implementierungsstrategien vorgestellt.

Einfache Heap-Verwaltung

In kleinen Systemen kann es sein, dass die Rechenleistung für eine vollständige Heap-Verwaltung nicht ausreicht. Kann man sich darauf beschränken, nur Speicher zu allokieren, aber nicht wieder freigeben zu müssen, ist eine einfache und effiziente Implementierung wie in Listing XX gezeigt möglich.

Der Allokationsvorgang ist sehr effizient, und der Speicher wird ohne Lücken optimal ausgenutzt. Die Größe der reservierten Speicherblöcke entspricht der angeforderten Größe, von eventuellen Ausrichtungskorrekturen auf Wortgrenzen, die manche Rechner verlangen, einmal abgesehen. Der Allokationsvorgang selbst ist als kritischer Abschnitt ausgeführt und wird durch die Funktionsaufrufe in den Zeilen 23 und 36 entsprechend geschützt.

Die Effizienz bei der Allokation wird erkauft mit dem Verzicht auf die Möglichkeit einer Deallokation. Die Verwaltungsstruktur erlaubt es nicht, unbenötigten Speicher wieder in den Heap einzugliedern.

```
1  /*
2   * Eine sehr einfache malloc()-Implementierung. Bei dieser Implemen-
3   * tierung kann der Speicher nicht wieder freigegeben werden.
4   */
5
6  /* Mit dieser Struktur kann der Heap auf eine Wortgrenze ausgerichtet
7   * werden.
8   */
9  static struct heapStructure {
10   unsigned INT16 someDummy;
11   unsigned INT8 heapBytes[ HEAPSIZE ];
12 } theHeap;
13
14 /* Zeigt auf nächstes freie Byte */
15 static size_t nextFreeByte = ( size_t ) 0;
16
17 /*-------------------------------------------------------------*/
18
19 void *osMalloc( size_t numberOfBytes )
20 {
21   void *reserviert = NULL;
22
23   enterCriticalSection();
24
25     /* Ist noch genügend Speicher da? */
26     if ((( nextFreeByte + numberOfBytes ) < HEAPSIZE ) &&
27         (( nextFreeByte + numberOfBytes ) > nextFreeByte ))
28     {
29       /* Gib den Zeiger auf das nächste freie Byte zurück,
30        * dann rücke Zeiger weiter bis Blockende vor
31        */
32       reserviert = &( theHeap.heapBytes[ nextFreeByte ] );
33       nextFreeByte = nextFreeByte + numberOfBytes;
34     }
35
36   exitCriticalSection();
37
38   return reserviert;
39 }
40 /*-------------------------------------------------------------*/
41
42 void osFree( void *pv )
43 {
44   /* Speicher kann nicht zurückgegeben werden.
45   ( void ) pv;
46 }
47 /*-------------------------------------------------------------*/
48
```

Bild 5-3 Einfache Allokationsroutine ohne Rückgabemöglichkeit

Heap-Verwaltung mit Allokationstabelle

In dieser Lösung wird der Heap in Blöcke gleicher Größe, z.B. 32 Byte unterteilt. Ein statisches Integer-Array verwaltet, welche Blöcke belegt und welche frei sind. Ist der Heap z.B. 100 Blöcke groß, muss das Integer-Array aus 100 Elementen bestehen.

Jedes Element des Integer-Arrays verweist auf die erste Adresse des ihm zugeordneten Blockes gemäß

```
blockStartAdresse = Offset + BlockGröße * ArrayIndex
```

Bild 5-3 zeigt eine Beispielstruktur mit 10 Elementen. Wenn der Heap komplett unbenutzt ist, enthält das erste Element eine 10, und das zehnte ebenfalls. Werden nun 40 Byte angefordert, rechnet die `malloc()`-Routine das auf zwei Blocks um. Dabei gehen 24 Byte verloren. Das bezeichnet man als interne Fragmentierung. Wenn die zwei Blöcke allokiert werden, wird in das letzte Element des Arrays eine 0 geschrieben, und in das vorletzte eine -2. Davor steht eine 8, und im ersten Element ebenfalls eine 8. Die 0 bedeutet, dass hier eine belegte Blockstrecke zuende ist, und die -2 bezeichnet den Beginn einer belegten Blockstrecke.

Bild 5-3 Mehrfacher Allokationsvorgang

Kommt nun eine weitere Anfrage z.B. für 80 Byte, werden drei weitere Blöcke belegt. Die `malloc()`-Routine sucht nach dem größten freien Block, und nimmt von dessen Ende drei Blöcke weg. Die entsprechende Blockstrecke wird mit -3 und 0 markiert, die verbleibende freie Blockstrecke mit 5 am Anfang und am Ende.

Bei dieser Art der Verwaltung liegt die Schwierigkeit darin, schnell eine geeignete freie Blockstrecke zu finden. Das könnte z.B. dadurch realisiert werden, dass das Integer-Array von vorne nach einer positiven Zahl

durchsucht wird, die der gewünschten Anzahl Blocks entspricht oder sie übersteigt. Eine solche Implementierung würde aber zu nicht-deterministischen Allokationszeiten führen und bei großen Speicherbereichen auch zeitaufwändig sein.
Deshalb legt man neben die Verwaltungsstruktur mit dem Integer-Array eine weitere Struktur an, in der die freien Blockstrecken nach Größe sortiert gehalten werden. Für die Sortierung, die bei der Belegung und Freigabe ablaufen muss, werden schnelle Sortieralgorithmen verwendet.

Bild 5-4 Freigabe von zuvor allokiertem Speicher

Bei der Rückgabe von Speicher ersetzt die `free()`-Funktion die negative Blockanzahl durch die entsprechende positive Zahl und die Null ebenfalls. Dann wird vor der positiven Zahl geschaut, ob dort ebenfalls eine positive Zahl steht. Wenn ja, werden die beiden Blockstrecken zusammengefasst zu einer größeren Blockstrecke. Dann wird hinter dem Element, dass die 0 beinhaltet hatte, ebenfalls geschaut, ob dort eine positive Zahl steht. Wenn ja, wird der dort beginnende freie Block mit dem gerade freigegebenen zusammengefasst. Diese Operationen des Zusammenfassens von kleinen freien Blöcken zu großen zusammenhängenden ist in dieser Struktur sehr effizient zu realisieren.

Blockierende und nicht blockierende Heap-Verwaltung

Die ANSI-C-Funktionen `malloc()` und `free()` bieten dem Programmierer keine Möglichkeit festzulegen, ob auf die Freigabe von Speicher gewartet werden soll, falls keiner zur Verfügung steht, oder ob mit einem entsprechenden Statuscode zurückgekehrt werden soll.

In dedizierten Systemen mit ihren beschränkten Ressourcen kann es schon einmal passieren, dass der gesamte Speicher gerade belegt ist, kurz darauf aber schon wieder freigegeben wird. Kommt in einem solchen Moment eine andere Task und benötigt Speicher, würde die Standard-Routine `malloc()` sofort mit einem Fehlercode zurückkehren, und der Anwendungsprogrammierer muss sich um die Behandlung des Fehlers kümmern. Manchmal ist es akzeptabel, wenn die Speicherzuteilung nicht sofort erfolgt und die Task blokkiert, bis wieder Speicher frei wird. Das würde den Programmierer entlasten und kann zu einer einfacheren und robusteren Programmstruktur führen.

Die meisten kommerziellen Echtzeitbetriebssysteme bieten eigene Implementierungen von `malloc()` und `free()` an, die um diese Funktion ergänzt sind. In RMOS z.B. sieht diese Funktion so aus:

Rm

In VxWorks sieht diese Funktion so aus:

Probleme bei der Heap-Verwaltung

Während die Stackverwaltung vom Compiler umgesetzt wird, wirkt bei einer Heap-Verwaltung der Programmierer maßgeblich mit. Das führt dazu, dass sich bei der Programmierung eine Reihe von sehr unangenehmen und z.T. schwer auffindbaren Fehlern einschleichen können.

Zugriff auf schon freigegebenen Speicher

Ein verbreiteter Fehler ist die zu frühe Freigabe von allokiertem Speicher mit `free()`. Die freigebende Instanz vergisst, dass eine andere Instanz noch einen Zeiger auf dieses Speicherbereich besitzt. Nach der Freigabe arbeitet alles weiter wie vorher. Ein Fehlverhalten tritt erst auf, wenn der deallokierte Speicher von einer anderen Instanz angefordert und belegt worden ist. Nun überschreiben sich die beiden Instanzen gegenseitig ihre Daten.

```
1  char* derSpeicher;
2
3  struct bigStruct {
4    INT16 wort1;
5    INT16 wort2;
6  }
7
8  main() {
9    task1();
10   task2();
11 }
12
13 void task1(){
14   derSpeicher = (char*) malloc(sizeOf(bigStruct));
15   if (derSpeicher == null) fehler(); /* Mit Ausstieg... */
16   machWas(derSpeicher);
17   free(derSpeicher);
18 }
19
20 void task2(){
21   char* lokal;
22   lokal = derSpeicher;
23   machWasAnderes(lokal);  /* Potentielles Problem */
24 }
```

Bild 5-5 Beispiel für Zugriff auf schon freigegebenen Speicher

Im Beispiel würde wahrscheinlich nie ein Problem auftreten, es sein denn, eine weitere Task würde parallel zu den vorhandenen arbeiten und ebenfalls Speicher allokieren wollen. In diesem Fall würde der von `task2()` unrechtmäßig benutzte Speicher anderweitig vergeben und es kommt zu einer Kollision.

Ein solcher Fehler ist nicht leicht zu finden, da es keinen zeitlichen Zusammenhang zwischen dem Fehlverhalten und der zu frühen Freigabe des allokierten Speichers gibt. Im Feld kann es u.U. mehrere Tage dauern, bis sich der Fehler bemerkbar macht, und nichts ist schwieriger zu finden als ein Fehler, der nur alle paar Tage oder Wochen einmal auftaucht.

Speicherlöcher

Der Programmierer kann leicht vergessen, zuvor mit `malloc()` allokierten Speicher wieder freizugeben. Geht nun der Zeiger auf diesen allokierten Speicher verloren, z.B. dadurch, dass die ihn besitzende Funktion beendet wird, gibt es im Programm keine Möglichkeit mehr, diesen Speicherbereich wieder nutzbar zu machen. Die Logik des Programms ist dadurch nicht betroffen, es arbeitet weiter wie vorher auch. Jedoch

sammeln sich im Laufe der Zeit diese verlorenen Speicherbereiche an, und es entstehen für die Software nicht mehr nutzbare Bereiche („Löcher“) im Speicher. Irgendwann steht kein dynamischer Speicher mehr zur Verfügung und die Software ist nicht mehr funktionsfähig.

```
1  char* derSpeicher;
2
3  struct bigStruct {
4    INT16 wort1;
5    INT16 wort2;
6  }
7
8  main() {
9    while (1) {
10      task1();
11   }
12 }
13
14 void task1(){
15   derSpeicher = (char*) malloc(sizeOf(bigStruct));
16   if (derSpeicher == null) fehler(); /* Mit Ausstieg... */
17   machWasMit(derSpeicher);
18   /* wir haben vergessen, wieder freizugeben */
19 }
```

Bild 5-6 Beispiel für die Entstehung eines Speicherlochs

Während kleine Speicherlöcher in Desktop-Systemen manchem akzeptabel erscheinen, sind sie in Serveranwendungen und in dedizierten Systemen völlig unakzeptabel.

Die gleiche Problematik taucht in C++ noch verschärft auf. Hier können zur Laufzeit Objekte angelegt und wieder gelöscht werden, die selbst ihre Daten allokieren und deallokieren. In der Regel legt man die Allokation von Speicher in den Konstruktor, und die Deallokation in den Destruktor. Es gibt aber nun Fälle, z.B. beim Auftreten von Ausnahmen, in denen der Destruktor nicht mehr ordnungsgemäß ausgeführt wird. Auch in einem solchen Fall kann es zu Speicherlöchern kommen. C++-Programme darf man deshalb nie ohne vorherige Tests mit Allokationscheckern freigeben.

Speicherfragmentierung

Die Allokationsalgorithmen in der Funktion `malloc()` führen dazu, dass der Heap-Speicher im Laufe der Zeit durch die Zuteilung unterschiedlich großer Anforderungen zerteilt oder fragmentiert wird. So entstehen auf dem Heap Speicherbereiche, die zu klein sind, um noch sinnvoll genutzt werden zu können und die

damit für die Anwendung verloren sind. Dieses Problem lässt sich in der Anwendungssoftware nicht korrigieren.

Dieses Problem der externen Speicherfragmentierung ist neben der Gefahr der Speicherlöcher der wesentliche Grund, weshalb erfahrene Programmierer von dedizierten Systemen weitgehend auf die dynamische Allokation und Deallokation von Speicher mit `new()` und `delete()` oder `malloc()` und `free()` verzichten, bzw. ihre eigenen, auf die Anwendung angepassten Varianten dieser Funktionen zur Verfügung stellen.

Bild 5-7 zeigt beispielhaft eine Allokation von 10 Byte auf einem Heap. In dieser Implementierung wird eine verbundene Liste (linked list) für die Verwaltung des freien und schon allokierten Heap-Speichers verwendet.

Bild 5-7 Erste Allokation von 10 Byte vom Heap

Bild 5-8 zeigt den gleichen Heap nach einer Reihe weiterer Allokationen. Auf der rechten Seite der Abbildung ist der Heap nach einer Deallokation eines Speicherblockes von 10 Byte dargestellt. Es steht nun ein Block von 15 Byte für weitere Allokationen zur Verfügung. Werden wieder 10 Byte angefordert, kann diese Anforderung bedient werden. Die verbleibenden 5 Byte können aber zu klein sein, um weitere Anforderungen zu bedienen und sind damit für die Anwendungssoftware verloren.

Bild 5-8 Entstehung der Fragmentierung

Das Problem der Speicherfragmentierung zur Laufzeit lässt sich deutlich reduzieren, wenn der Heap nicht aus einer Menge Blocks gleicher Größe besteht, sondern in eine Reihe von Teilmengen mit Blocks unterschiedlicher Größe aufgeteilt wird. Bei der Allokation wird immer nur ein Block zurückgegeben, aber nie eine zusammenhängende Menge an Blocks. Benötigt der Anwendungsprogrammierer nur wenige Byte, bedient die `malloc()`-Routine diese Anforderung aus der Menge Blocks mit kleiner Blockgröße, z.B. 16 Byte. Benötigt der Anwender z.B. 250 Byte, bedient die `malloc()`-Routine dies aus der Menge Blocks mit einer Blockgröße von 256 Byte. Wird noch mehr benötigt, müssen entweder noch Mengen von noch größeren Blocks reserviert werden, oder man fällt wieder auf die Strategie der Zusammenfassung von Blocks zurück, mit dem Nachteil einer möglichen Fragmentierung.

Multitasking

In Multitasking-Systemen muss jede Task ihren eigenen Stack haben, es ist aber nicht notwendig, dass sie einen eigenen Heap besitzt. Einige RTOS unterstützen verschiedene Heaps (manche nennen das eine „Region“), die einzelnen Tasks zugeteilt werden können. Wird ein Heap für mehrere Tasks verwendet, müssen die Funktionen `malloc()` und `free()` reentrant programmiert sein bzw. es müssen die vom Betriebssystem zur Verfügung gestellten Routinen verwendet werden.

Ein gemeinsamer Heap ist auch hilfreich, wenn Daten zwischen Tasks ausgetauscht werden sollen. Hier muss man jedoch eine Regel aufstellen, wer für den allokierten Speicher zu welchem Zeitpunkt verantwortlich ist, und wer für die Rückgabe zuständig ist, um Speicherlöcher oder Zeigerfehler zu vermeiden.

Bild 5-9 Heap-Struktur mit mehreren Blockgrößen

5.6.4 Zusammenfassung Speicherverwaltung

Generell ist zu berücksichtigen, dass auf dedizierten Systemen dynamische Speicher-Allokation und -Deallokation zu Problemen führen kann. Die Menge des zur Verfügung stehenden Speichers ist bei dedizierten Systemen meist festgelegt. In der Regel gibt es keinen virtuellen Speicher, und zumeist laufen nur vorher bekannte Anwendungen auf der Hardware, deren Ressourcen-Bedarf a priori bekannt sein muss.

Es gibt natürlich trotzdem hier und dort Bedarf für eine dynamische Speicherallokation und -Deallokation. Sie sollte jedoch mit Argwohn betrachtet werden und an möglichst wenigen Orten im System zusammengezogen werden. So können z.B. alle Objekte in einem Initialisierungsteil mit einer „Fabrik“ erzeugt werden. Code, der mit `new()` und `delete()` oder `malloc()` und `free()` durchzogen ist, muss als problematisch und fehleranfällig betrachtet werden.

Neben der Fehleranfälligkeit der Operationen zur dynamischen Allokation und Deallokation von Speicher führt vor allem ein Effekt in einem dedizierten System zu Problemen: die Fragmentierung von Speicher.

Abhilfe schaffen hier nur spezielle, auf die Anwendung zugeschnittene Speicherverwaltungsroutinen, oder der Verzicht auf dynamische Allokation zur Laufzeit.

Aufgaben

1. Schreiben Sie ein Programm mit dem Betriebssystem RMOS, dass mit Hilfe von vier Tasks das Problem der Speicherfragmentierung demonstriert. Dazu soll jeder Task Speicherblöcke mit zufälliger Größe allokieren und nach einer zufälligen Zeit wieder deallokieren. Jeder Task soll gleichzeitig immer mindestens fünf Speicherblöcke allokiert haben. Die Summe des allokierten Speichers aller Tasks soll nie größer als 80% des zur Verfügung stehenden Systemspeichers betragen. Messen Sie die Zeit bis zum ersten „Out of Memory Error“ und geben Sie den in diesem Moment insgesamt allokierten Speicher plus die Menge des gerade angeforderten Speichers auf der Konsole aus. Dann beenden sie alle Tasks.
2. Für eine Heap-basierte Speicherverwaltung stehen 100 Byte RAM zur Verfügung. Eine Anwendung benutzt diesen Speicher nun wie folgt:
    1. Sie fordert 20 Byte an (Block 1)
    2. Sie fordert zweimal hintereinander 10 Byte an (Block 2 und 3)
    3. Sie fordert 20 Byte an (Block 4)
    4. Sie fordert 20 Byte an (Block 5)
    5. Sie gibt Block 4 zurück
    6. Sie fordert 30 Byte an (Block 6)

    Wieviel Speicher steht vor der Anforderung von Block 6 noch zur Verfügung. Kann die Anfrage befriedigt werden?

Anhang A

A.1 Installation der Entwicklungsumgebung

1. Wechseln Sie auf der Installations-CD in das Verzeichnis „Metrowerks Codewarrior“
2. Führen Sie die Datei „CW12_V3_1.exe“ aus.
3. Als Zielpfad für die Installation geben Sie „C:\Programme\Metrowerks\CodeWarrior CW12_V3.1“ an.
4. Wählen Sie „typical“ als Installationstyp aus.
5. Stellen Sie das File Extension Mapping wie in Abb. A-1 ein.
6. Antworten Sie auf die Frage nach einem Update und der Installation eines BDM-Treibers mit „Nein“.
7. Booten Sie Ihren Rechner neu.
8. Führen Sie die Datei „CW12_V3_1_PE_V2.95_SP.exe“ aus. Ignorieren Sie die Aufforderung zum Entfernen der vorhergehenden Processor Expert Version.
9. Kopieren Sie die Datei „license.dat“ aus dem Metrowerks Codewarrior Verzeichnis nach „C:\Programme\Metrowerks\CodeWarrior CW12_V3.1\license.dat“.
10. Kopieren Sie die Datei „hc12.ini“ nach „C:\Programme\Metrowerks\CodeWarrior CW12_V3.1\prog\hc12.ini“.

11. Wenn Sie Zugang zu einem Board haben, verbinden Sie dieses an der Buchse am LCD-Display mit Hilfe des beigefügten seriellen Kabels mit der COM1-Schnittstelle Ihres PCs. Schließen Sie die Versorgungsspannung an.
12. Stellen Sie sicher, dass aller Jumper und Schalter, insbesondere der DIP-Schalter SW7 rechts unten richtig eingestellt sind. Wenn die Versorgungsspannung an das Board gelegt wird, muss die LED EVB rechts unten leuchten.
13. Kopieren Sie den Ordner „Beispiele\IDE-Test“ an eine geeignete Stelle in Ihrem persönlichen Verzeichnis (z.B. „Eigene Dateien\rt2\IDE-Test“).
14. Wechseln Sie in das „Start“-Menü von Windows und starten Sie die CodeWarrior IDE.
15. Gehen Sie mit „File“-“Open“ in das Verzeichnis eben angelegte Verzeichnis „IDE-Test“.
16. Wählen Sie die Projektdatei „IDE-Test.mcp“ aus.
17. Wenn Sie ein Board haben, drücken Sie den Reset-Knopf des Boards. Ansonsten stellen Sie in der Entwicklungsumgebung das Target oben rechts von „Monitor“ auf „Simulator.
18. Gehen Sie in der Entwicklungsumgebung im Menü oben auf „Project“-“Debug“.
19. Steppen Sie im Debugger durch den Code. Wenn Sie ein Board haben, sollte auf der Siebensegmentanzeige eine Nachricht aufleuchten. Das können Sie im Simulator leider nicht sehen.

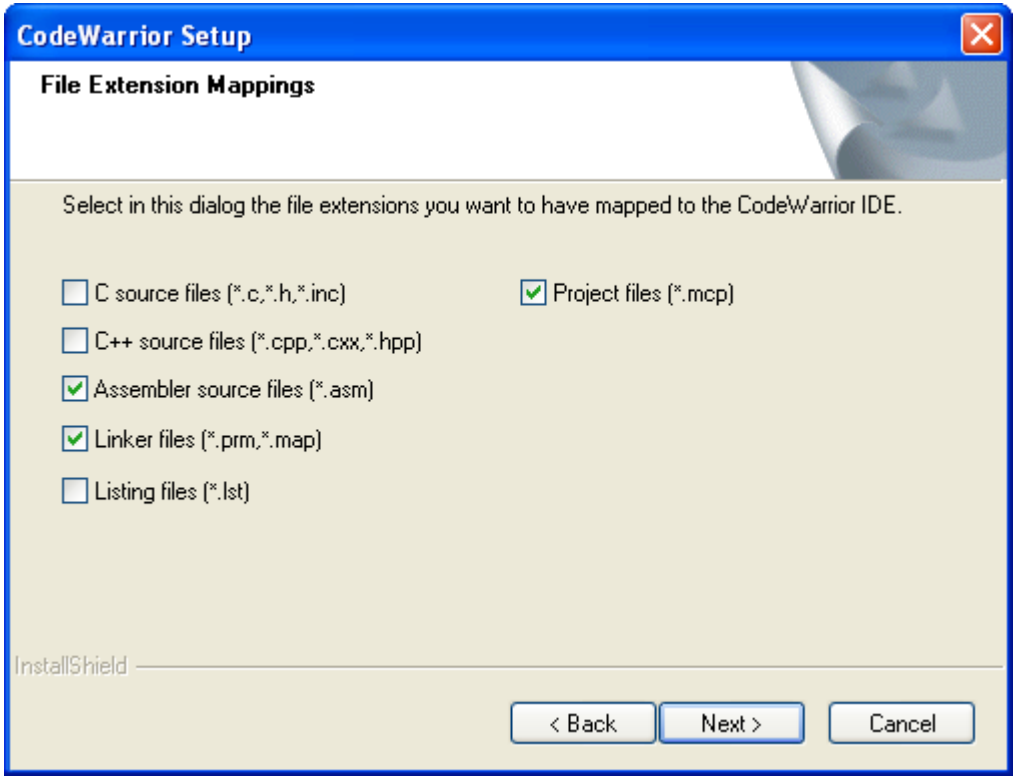


Abbildung A-1: Stellung des File Extension Mappings

Anhang B

B.1 Hardware für die Laborübungen

Für die Laborübungen wird eine Hardware verwendet, die mit einem 16-Bit Freescale-Prozessor vom Typ MC9S12DP256B ausgestattet ist. Das reichhaltig ausgestattete Board „Dragon12“ kann unter http://www.evbplus.com für ca. 150,00 Euro (USD 139,00 plus 19% Zollgebühr) käuflich erworben werden. Ein Teil der Versuche und Übungen kann auch ohne das Board mit dem Simulator der Entwicklungsumgebung durchgeführt werden.

Zum Betrieb des Boards ist noch ein 9-Volt Netzteil mit mindestens 6 Watt erforderlich. Für die Verbindung mit dem Entwicklungsrechner benötigt man einen RS-232-Anschluss.

Für die Laborversuche haben wir den mitgelieferten Debug-Monitor durch einen Monitor ersetzt, der mit der Metrowerks-Entwicklungsumgebung zusammenarbeitet. Die ursprünglichen Debug-12-Kommandos funktionieren also nicht mehr.

B.2 Allgemeine Hinweise

Das Herunterladen von Software auf den Zielrechner ist in der Regel problemlos möglich. Man muss aber sicher stellen, dass der Debug-Monitor aktiv ist. Es empfiehlt sich deshalb, vor dem Herunterladen den Reset-Knopf des Zielsystems zu betätigen.

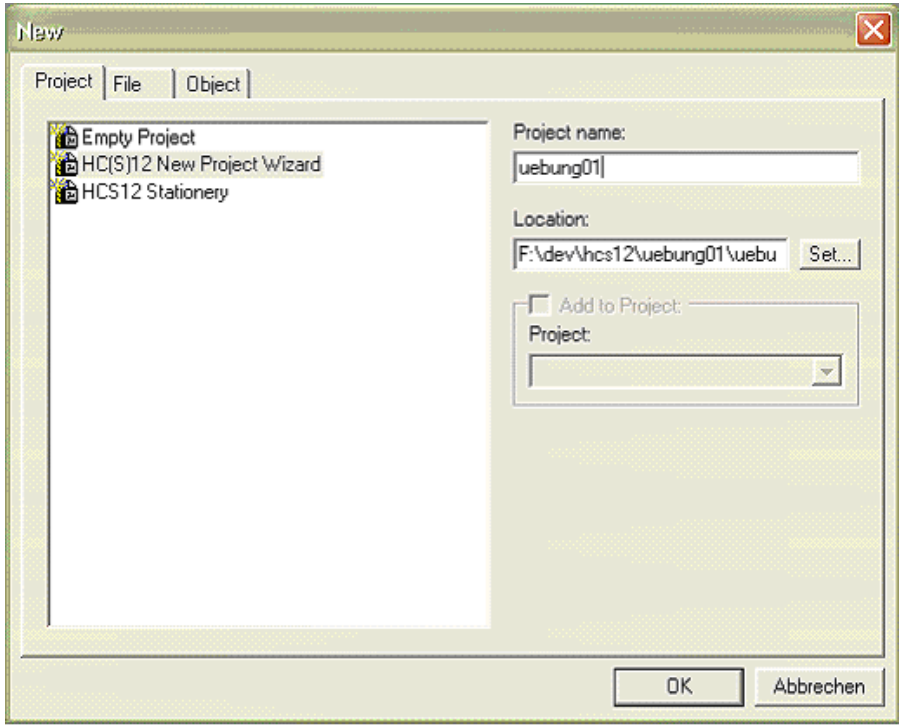
Das Board muss im EVB-Modus konfiguriert sein, damit es mit dem Debugger zusammen arbeitet. Das bedeutet, dass SW7 so eingestellt sein muss, das die linke LED (EVB) rechts unten in der Ecke leuchtet.

Der Monitor stellt die Bus-Taktfrequenz des Prozessors auf 24 MHz ein. Das sollte auch nicht geändert werden, da sonst die Anbindung des Debuggers über die serielle Schnittstelle nicht mehr richtig arbeitet.

Hat man keinen Erfolg beim Herunterladen der Software, ohne dass eine Fehlermeldung erscheint, hat man wahrscheinlich statt „Monitor“ „Simulator“ als Debug-Ziel gewählt.

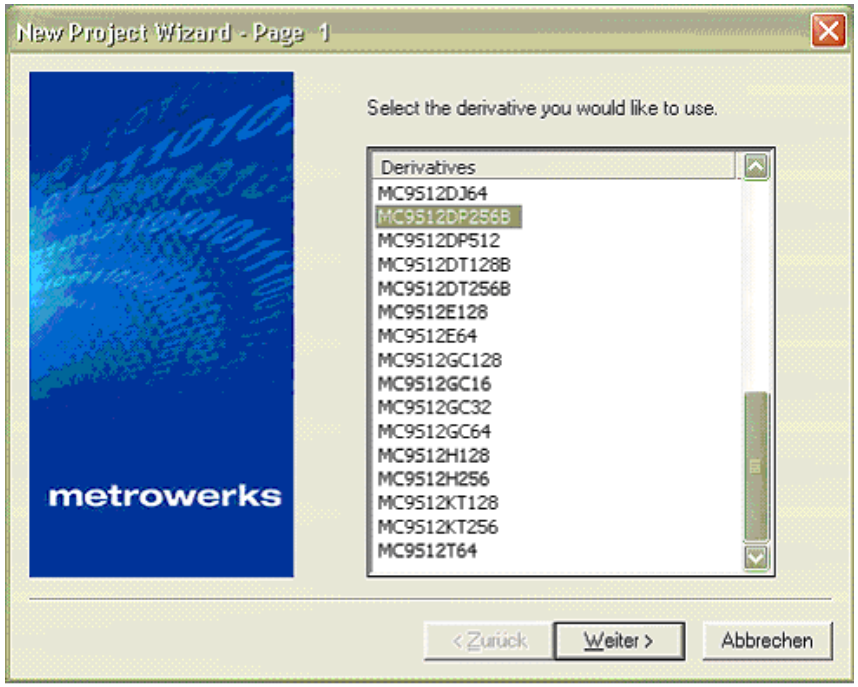
B.3 Anlegen eines neuen Projekts

Neue Projekte kann man anlegen, indem man ein existierenden kopiert oder über den Assistenten ein neues erzeugt. Hier ist kurz die Vorgehensweise mit Hilfe des Assistenten beschrieben. Über <File> und <New> wird der Assistent gestartet.



Man vergibt einen geeigneten Projektnamen und einen Ort, wie die Projektdateien gespeichert werden sollen.

Als nächstes ist der richtige Prozessortyp auszuwählen. Auf dem Labor-Board läuft ein MC9S12DP256B.



Als nächstes wählen wir, ob wir ein Assembler-Projekt oder ein C-Projekt anlegen wollen. Es können auch gemischte Projekte angelegt werden. Für die Option „C++“ haben wir leider keine Lizenz. Wir wählen für unser Beispiel ein Assemblerprojekt, und zwar ein sogenanntes „relokierbares“. Damit können wir Assemblerprogramme schreiben, bei denen über den Linker festgelegt wird, wohin im Speicher sie später gelegt werden. Außerdem können zu dem Projekt so mehrere Assemblerquelldateien gehören, ohne dass wir uns über die Anordnung zu viel Gedanken machen müssen.

B.4 Peripherie

Im Labor wird nur ein Teil der zur Verfügung stehenden Peripherie verwendet. In den folgenden Abschnitten sind die wichtigsten Elemente beschrieben. Weitere Informationen erhält man durch Studium des Schaltbildes, das im Dokumentationsset der Vorlesung zur Verfügung steht.

B.4.1 LED-Zeile

Es steht eine Zeile mit acht roten LEDs zur Verfügung. Diese sind am Port B des Prozessors angeschlossen, und zwar so, dass ein positives Signal auf den Port gegeben werden muss, damit die LEDs eingeschaltet werden. Jede LED ist mit einem Bit des Ports B verbunden; die am weitesten links stehende LED ist mit Bit 0 verbunden.

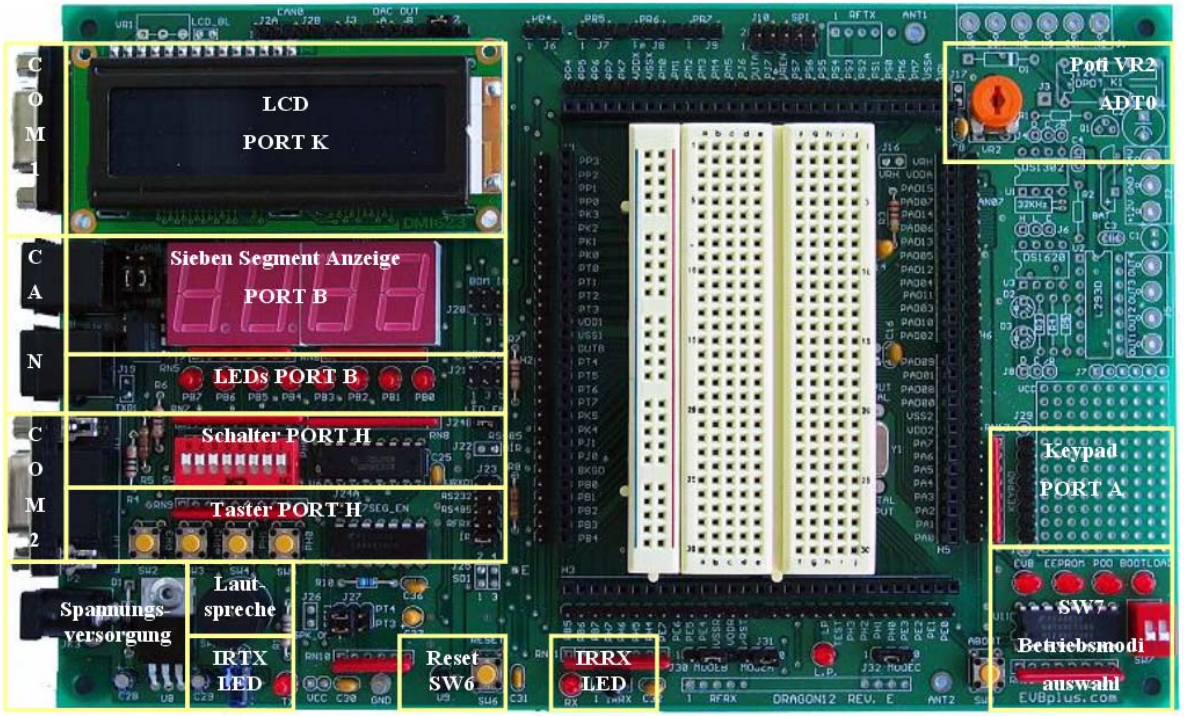


Abbildung B-1: Das im Labor verwendete Dragon12-Board von Wytec

Port B wird zweifach verwendet: einmal für die LED-Zeile und dann noch für die LED-Siebensegment-Anzeige. Um die LED-Zeile zu aktivieren und nicht das Siebensegment-Display, muss Bit 1 von Port J auf null gesetzt werden.
Beispiel in C:

```
DDRJ_DDRJ1 = 1; /* Datenrichtungsregister auf Ausgang schalten */

PTJ_PTJ1 = 0;   /* Port J, Bit 1 auf null setzen */

DDRB = 0xff;    /* Datenrichtungsregister auf Ausgang schalten */

PORTB = 0xff;   /* alle LEDs einschalten */
```

Die Namen der Register können in Assembler und C leicht unterschiedlich benannt sein; sie können sich auch etwas von den Namen in der Freescale-Dokumentation unterscheiden. Entscheidend sind bei Assemblerprogrammierung die Namen in der Datei `mc9s12dp256.inc`. Die Basisadressen für die einzelnen I/O-Registerblöcke sind in Tabelle B.1aufgeführt. Die Adressen der einzelnen Register innerhalb eines I/O-Blocks ergeben sich aus den Basisadressen plus den im jeweiligen Dokument aufgeführten Register-Offsetadressen.
Alle Register stehen mit ihren Adressen schon in der oben erwähnten Include-Datei, man muss sie also nicht selbst definieren. Der Einfachheit halber folgt hier ein unvollständiger Auszug:

```
; ###################################################################
;     Filename  : mc9s12dp256.inc
; ###################################################################

;*** Memory Map and Interrupt Vectors
;*************************************
RAMStart:           equ   $00001000
RAMEnd:             equ   $00003FFF
ROM_C000Start:      equ   $0000C000
ROM_C000End:        equ   $0000FF7F
Vportp:             equ   $0000FF8E
Vporth:             equ   $0000FFCC
Vportj:             equ   $0000FFCE
```

Tabelle B.1: Device Memory Map

Address	Module	Size (Bytes)
$0000 - $0017	CORE (Ports A, B, E, Modes, Inits, Test)	24
$0018 - $0019	Reserved	2
$001A - $001B	Device ID register (PARTID)	2
$001C - $001F	CORE (MEMSIZ, IRQ, HPRIO)	4
$0020 - $0027	Reserved	8
$0028 - $002F	CORE (Background Debug Mode)	8
$0030 - $0033	CORE (PPAGE, Port K)	4
$0034 - $003F	Clock and Reset Generator (PLL, RTI, COP)	12
$0040 - $007F	Enhanced Capture Timer 16-bit 8 channels	64
$0080 - $009F	Analog to Digital Converter 10-bit 8 channels (ATD0)	32
$00A0 - $00C7	Pulse Width Modulator 8-bit 8 channels (PWM)	40
$00C8 - $00CF	Serial Communications Interface 0 (SCI0)	8
$00D0 - $00D7	Serial Communications Interface 0 (SCI1)	8
$00D8 - $00DF	Serial Peripheral Interface (SPI0)	8
$00E0 - $00E7	Inter IC Bus	8
$00E8 - $00EF	Byte Data Link Controller (BDLC)	8
$00F0 - $00F7	Serial Peripheral Interface (SPI1)	8
$00F8 - $00FF	Serial Peripheral Interface (SPI2)	8
$0100- $010F	Flash Control Register	16
$0110 - $011B	EEPROM Control Register	12
$011C - $011F	Reserved	4
$0120 - $013F	Analog to Digital Converter 10-bit 8 channels (ATD1)	32
$0140 - $017F	Motorola Scalable Can (CAN0)	64
$0180 - $01BF	Motorola Scalable Can (CAN1)	64
$01C0 - $01FF	Motorola Scalable Can (CAN2)	64
$0200 - $023F	Motorola Scalable Can (CAN3)	64
$0240 - $027F	Port Integration Module (PIM)	64
$0280 - $02BF	Motorola Scalable Can (CAN4)	64
$02C0 - $03FF	Reserved	320
$0000 - $0FFF	EEPROM array	4096
$1000 - $3FFF	RAM array	12288
$4000 - $7FFF	Fixed Flash EEPROM	16384
$8000 - $BFFF	Flash EEPROM Page Window	16384
$C000 - $FFFF	Fixed Flash EEPROM	16384

```
Vatd1:              equ   $0000FFD0
Vatd0:              equ   $0000FFD2
Vtimovf:            equ   $0000FFDE
Vtimch7:            equ   $0000FFE0
Vtimch1:            equ   $0000FFEC
Vtimch0:            equ   $0000FFEE
Vrti:               equ   $0000FFF0
Virq:               equ   $0000FFF2
Vxirq:              equ   $0000FFF4
Vswi:               equ   $0000FFF6
Vtrap:              equ   $0000FFF8
Vcop:               equ   $0000FFFA
Vreset:             equ   $0000FFFE
;
;*** PORTAB - Port AB Register; 0x00000000 ***
PORTAB:             equ    $00000000

;*** PORTA - Port A Register; 0x00000000 ***
PORTA:              equ    $00000000

;*** PORTB - Port B Register; 0x00000001 ***
PORTB:              equ    $00000001

;*** DDRAB - Port AB Data Direction Register; 0x00000002 ***
DDRAB:              equ    $00000002
;*** DDRA - Port A Data Direction Register; 0x00000002 ***
DDRA:               equ    $00000002
;*** DDRB - Port B Data Direction Register; 0x00000003 ***
DDRB:               equ    $00000003

;*** PORTK - Port K Data Register; 0x00000032 ***
PORTK:              equ    $00000032

; bit numbers for user in BCLR, BSET, BRCLR and BRSET
PORTK_BIT0:         equ    0                                    ; Port K Bit 0
PORTK_BIT1:         equ    1                                    ; Port K Bit 1
PORTK_BIT2:         equ    2                                    ; Port K Bit 2
```

```
PORTK_BIT3:         equ    3                                    ; Port K Bit 3
PORTK_BIT4:         equ    4                                    ; Port K Bit 4
PORTK_BIT5:         equ    5                                    ; Port K Bit 5
PORTK_BIT7:         equ    7                                    ; Port K Bit 7
; bit position masks
mPORTK_BIT0:        equ    %00000001                           ; Port K Bit 0
mPORTK_BIT1:        equ    %00000010                           ; Port K Bit 1
mPORTK_BIT2:        equ    %00000100                           ; Port K Bit 2
mPORTK_BIT3:        equ    %00001000                           ; Port K Bit 3
mPORTK_BIT4:        equ    %00010000                           ; Port K Bit 4
mPORTK_BIT5:        equ    %00100000                           ; Port K Bit 5
mPORTK_BIT7:        equ    %10000000                           ; Port K Bit 7

;*** DDRK - Port K Data Direction Register; 0x00000033 ***
DDRK:               equ    $00000033

;*** SYNR - CRG Synthesizer Register; 0x00000034 ***
SYNR:               equ    $00000034

;*** REFDV - CRG Reference Divider Register; 0x00000035 ***
REFDV:              equ    $00000035

;*** CRGFLG - CRG Flags Register; 0x00000037 ***
CRGFLG:             equ    $00000037

;*** CRGINT - CRG Interrupt Enable Register; 0x00000038 ***
CRGINT:             equ    $00000038

;*** CLKSEL - CRG Clock Select Register; 0x00000039 ***
CLKSEL:             equ    $00000039

;*** PLLCTL - CRG PLL Control Register; 0x0000003A ***
PLLCTL:             equ    $0000003A

;*** RTICTL - CRG RTI Control Register; 0x0000003B ***
RTICTL:             equ    $0000003B

;*** ARMCOP - CRG COP Timer Arm/Reset Register; 0x0000003F ***
```

```
ARMCOP:             equ    $0000003F

;*** TIOS - Timer Input Capture/Output Compare Select; 0x00000040 ***
TIOS:               equ    $00000040

;*** OC7M - Output Compare 7 Mask Register; 0x00000042 ***
OC7M:               equ    $00000042

;*** OC7D - Output Compare 7 Data Register; 0x00000043 ***


;*** PTH - Port H I/O Register; 0x00000260 ***
PTH:                equ    $00000260

;*** PTIH - Port H Input Register; 0x00000261 ***
PTIH:               equ    $00000261

;*** DDRH - Port H Data Direction Register; 0x00000262 ***
DDRH:               equ    $00000262

;*** RDRH - Port H Reduced Drive Register; 0x00000263 ***
RDRH:               equ    $00000263

;*** PERH - Port H Pull Device Enable Register; 0x00000264 ***
PERH:               equ    $00000264

;*** PPSH - Port H Polarity Select Register; 0x00000265 ***
PPSH:               equ    $00000265

;*** PIEH - Port H Interrupt Enable Register; 0x00000266 ***
PIEH:               equ    $00000266

;*** PIFH - Port H Interrupt Flag Register; 0x00000267 ***
PIFH:               equ    $00000267

;*** PTJ - Port J I/O Register; 0x00000268 ***
PTJ:                equ    $00000268

;*** PTIJ - Port J Input Register; 0x00000269 ***
```

```
PTIJ:                equ    $00000269

;*** DDRJ - Port J Data Direction Register; 0x0000026A ***
DDRJ:                equ    $0000026A

;*** RDRJ - Port J Reduced Drive Register; 0x0000026B ***
RDRJ:                equ    $0000026B

;*** PERJ - Port J Pull Device Enable Register; 0x0000026C ***
PERJ:                equ    $0000026C

;*** PPSJ - PortJP Polarity Select Register; 0x0000026D ***
PPSJ:                equ    $0000026D

;*** PIEJ - Port J Interrupt Enable Register; 0x0000026E ***
PIEJ:                equ    $0000026E

;*** PIFJ - Port J Interrupt Flag Register; 0x0000026F ***
PIFJ:                equ    $0000026F
```

B.4.2 Schalter und Taster

Auf dem Board stehen eine Reihe von Schaltern und Tastern zur Verfügung. Die für die Anwendungsprogrammierung interessanten sind am Port H angeschlossen. Die ersten 4 Elemente des DIP-Schalters (PH0 bis PH3) sind mit den vier Tastern SW5 bis SW2 parallel geschaltet. Damit die Taster funktionieren, müssen die DIP-Schalter im Zustand „off“ sein, also in Richtung LCD geschoben.

Port H ist ein weitgehend programmierbarer Port, der auch Interrupts generieren kann. Sind Interrupts aktiviert, muss man auch eine Interrupt-Routine installiert haben, sonst stürzt der Rechner auf Tastendruck hin ab.

Eine umfangreiche Beschreibung der Funktionalität des Ports H findet sich im Dokument „003-Port Integration Module“. In C würde man den Taster SW5 so abfragen:

```
...
DDRH = 0x00; /* nur zur Initialisierung, Port H als Eingang schalten */
...
```

```
SW5 = ~(PTH & 0x01);  /* Gedrückt, wenn SW5 == 1 */
...
```

B.4.3 Siebensegment-Anzeige

Die Siebensegment-Anzeige ist parallel zu den acht roten LEDs an Port B angeschlossen. Die Ansteuerung erfolgt in einem sogenannten Zeitmultiplexverfahren. Es können immer nur die Segmente eines Elements angesteuert werden. Über Port P wird eingestellt, welches Element aktiv sein soll. Für das aktive Element muss das zugehörige Bit von Port P auf 0 gesetzt werden. In C würde das so aussehen:

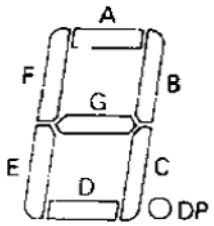
```
...
DDRJ_DDRJ1 = 1;  /* LED-Zeile deaktivieren */
PTJ_PTJ1    = 1;

DDRP = 0xff;   /* Port P auf Ausgang schalten */

PTP_PTP0 = 0; /* Linke Siebensegment-Anzeige einschalten */
PTP_PTP1 = 1; /* Alle anderen ausschalten */
PTP_PTP2 = 1;
PTP_PTP3  = 1;
...
```

Die Anzeigeelemente sind vertauscht montiert. Es ist allerdings durch die Verdrahtung auf dem Board sichergestellt, dass die Segmente gleich angesteuert werden, man muss also die Vertauschung beim Programmieren nicht berücksichtigen. Die Zuordnung der Segmente zu den Bits von Port B für die beiden linken Elemente ist in Abbildung B-2 dargestellt. Die beiden rechten Elemente kann man genau so ansteuern; Port B Bit 0 ist in diesem Fall z.B. mit Segment D verbunden.

B.4.4 LCD 16x2

Auf dem Board befindet sich ein zweizeiligen Liquid Crystal Display mit einer Zeilenlänge von 16 Zeichen. Das LCD besitzt selbst einen kleinen Mikrocontroller, der mit dem Controller des Laborboards über den

Port K kommuniziert. Das LCD wird im 4-Bit-Betrieb benutzt; es sind nur vier Datenleitungen mit dem Labor-Controller verbunden.




```
A  =  PORTB_BIT0
B  =   PORTB_BIT1
C  =   PORTB_BIT2
D  =   PORTB_BIT3
E  =   PORTB_BIT4
F  =   PORTB_BIT5
G  =   PORTB_BIT6
DP=   PORTB_BIT7
```

Abbildung B-2: Zuordnung der von Port B auf die Segmente der Siebensegment-Anzeige

Der LCD-Controller ist vom weit verbreiteten Typ HD 44780. Das entsprechende Datenblatt finden Sie in den Unterlagen. Die Verbindung zwischen Labor-Controller und LCD ist in Tabelle B.2 dargestellt. Es ist zu beachten, dass manche Instructionen an den LCD-Controller Wartezeiten erfordern.

Tabelle B.2: Anschluss des LCD an Port K des Controllers

Port K Bit	LCD Funktion
Bit 7	Nicht benutzt
Bit 6	Nicht benutzt
Bit 5	DB7 Data Bus

Tabelle B.2: Anschluss des LCD an Port K des Controllers

Port K Bit	LCD Funktion
Bit 4	DB6 Data Bus
Bit 3	DB5 Data Bus
Bit 2	DB4 Data Bus
Bit 1	EN - Enable Control Signal
Bit 0	RS - Register select (1=data, 0=instruction)

B.4.5 A/D-Wandler und Potentiometer

Auf dem Labor-Board steht ein Potentiometer zur Verfügung. Dieses ist mit dem Eingang AD07 des Analog-Digitalwandlers ATD0 verbunden. Die Bedeutung der einzelnen Register der A/D-Wandler ist im Dokument „008-AnalogToDigital“ beschrieben.

Die A/D-Wandler können im Interrupt-Modus oder Polling-Modus betrieben werden. Im Interrupt-Modus sieht die Verwendung in C z.B. so aus:

```
unsigned int  ADC_Data;/* globale Variable zum Austausch des */
               /* des gemessenen Wertes              */

/* Die Interrupt-Routine, wird selbständig von Hardware aufgerufen */
interrupt void ADC_ISR(void) {
  ADC_Data          = ATD0DR0;
  ATD0CTL2_ASCIF    = 0;
  ATD0CTL5_CA       = 1;
  ATD0CTL5_CB       = 1;
  ATD0CTL5_CC       = 1;
  ATD0CTL5_MULT     = 0;
  ATD0CTL5_SCAN     = 0;
  ATD0CTL5_DSGN     = 0;
  ATD0CTL5_DJM      = 1;
```

```
}

/* Initialisierung des Wandlers */
void ADC_Init(void){
  ATD0CTL2_ASCIF    = 0;
  ATD0CTL2_ASCIE    = 1;   /* Enable Interrupt */
  ATD0CTL2_ETRIGE   = 0;
  ATD0CTL2_ETRIGP   = 0;
  ATD0CTL2_ETRIGLE  = 0;
  ATD0CTL2_AWAI     = 0;
  ATD0CTL2_AFFC     = 0;
  ATD0CTL2_ADPU     = 1;   /* A/D - Wandler Einschalten */
  ATD0CTL3_FRZ0     = 0;
  ATD0CTL3_FRZ      = 0;
  ATD0CTL3_FIFO     = 0;
  ATD0CTL3_S1C      = 1;   /* Eine Wandlung pro Sequenz; Ergebnis in ATD0DR0 */
  ATD0CTL3_S2C      = 0;
  ATD0CTL3_S4C      = 0;
  ATD0CTL3_S8C      = 0;
  ATD0CTL4_PRS0     = 1;   /* 24MHz Clock */
  ATD0CTL4_PRS1     = 0;
  ATD0CTL4_PRS2     = 1;
  ATD0CTL4_PRS3     = 0;
  ATD0CTL4_PRS4     = 0;
  ATD0CTL4_SMP0     = 0;
  ATD0CTL4_SMP1     = 0;
  ATD0CTL4_SRES8    = 0;
  ATD0CTL5_CA       = 1;  /* AD7 Eingang - Potentiometer */
  ATD0CTL5_CB       = 1;
  ATD0CTL5_CC       = 1;
  ATD0CTL5_MULT     = 0;
  ATD0CTL5_SCAN     = 0;
  ATD0CTL5_DSGN     = 0;
  ATD0CTL5_DJM      = 1;
}
```

B.4.6 Lautsprecher

Auf dem Board ist ein kleiner Lautsprecher montiert. Dieser ist mit dem Bit 5 des Port T verbunden. Durch Hin- und Herschalten mit einer entsprechenden Frequenz lässt sich so ein Ton erzeugen. Port T Bit 5 lässt sich sehr gut durch die Timer-Hardware direkt ansteuern.

B.5 Codewarrior-Simulator

Die Metrowerks-Entwicklungsumgebung beinhaltet einen Simulator für den Freescale-Rechner, der auf dem Dragon12-Board verwendet wird. Der Simulator erlaubt das Ausführen von Programmen ohne die Hardware. Es stehen allerdings nicht alle Ein-/Ausgabeelemente zur Verfügung, und die zur Verfügung stehenden verhalten sich u.U. anders als die Hardware auf dem Dragon12-Board

Die Komponenten können im Simulator über das Menü „Components“ erreicht werden. Hilfreich kann vor allem die Komponente „Led“ sein. Um diese Komponente mit dem Port B zu verknüpfen, muss man mit der rechten Maustaste auf die Komponente klicken und in das Setup-Fenster folgenden String eingeben:

TargetObject.#1

Die „1“ steht dabei für die Adresse des Ports B (0x001). Entsprechend kann man auch andere Ports anschließen.

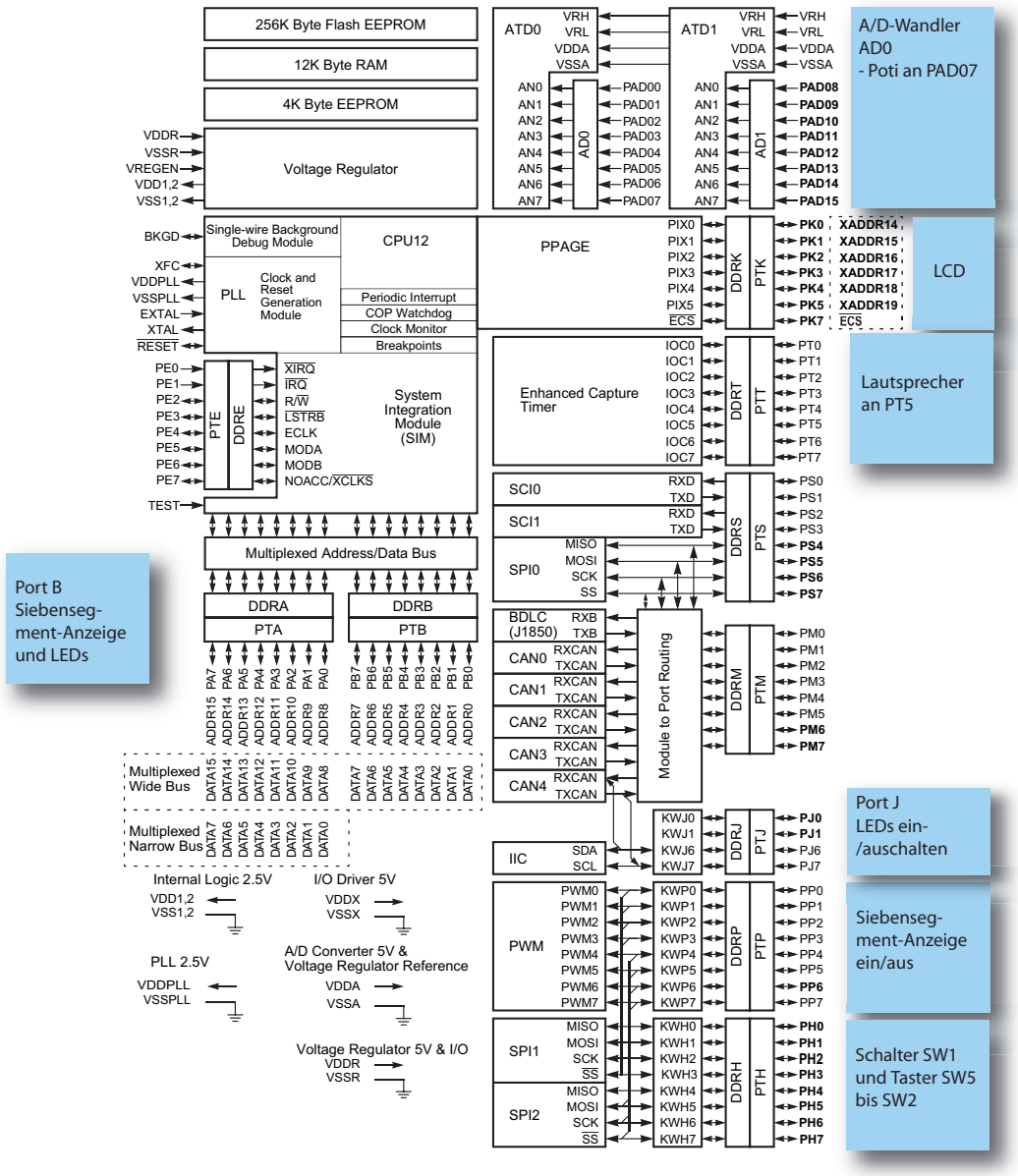


Abbildung B-3: Device-Übersicht

Index