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

- Mo, 11:15 bis 12:45 in Raum F1.216, Mi. 9:30 bis 11:00 in Raum F1.311
- Teilweise Zusatztermine Mo, 13:30 bis 15:00 Uhr, bitte freihalten
- 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*.

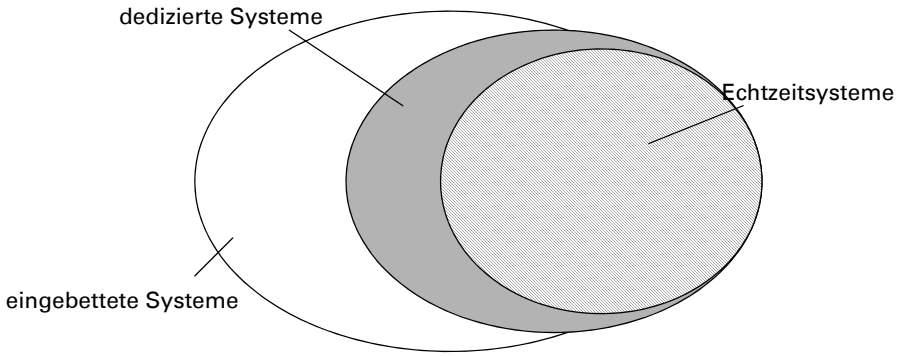


Bild 1.1: Abgrenzung eingebettete, dedizierte und Echtzeitsysteme

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

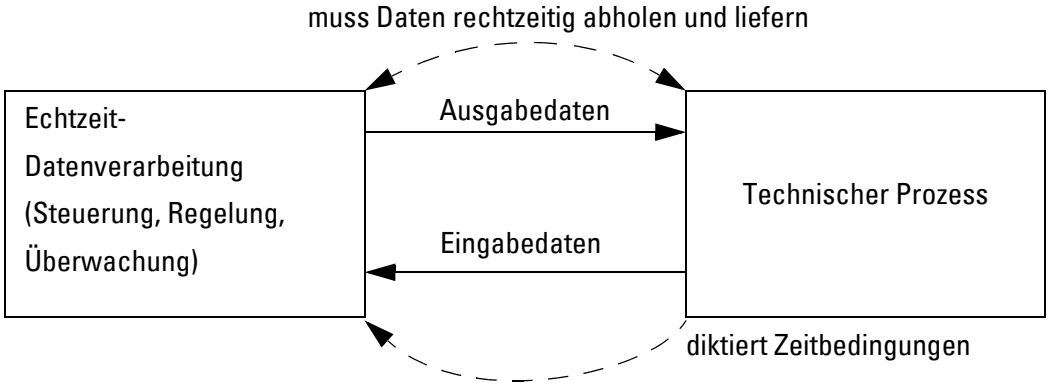
- Nicht-Echtzeitsysteme: *logische Korrektheit* bedeutet Korrektheit
- Echtzeitsysteme: *logische Korrektheit plus zeitliche Korrektheit* bedeutet Korrektheit

1.7.1 Rechtzeitigkeit

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

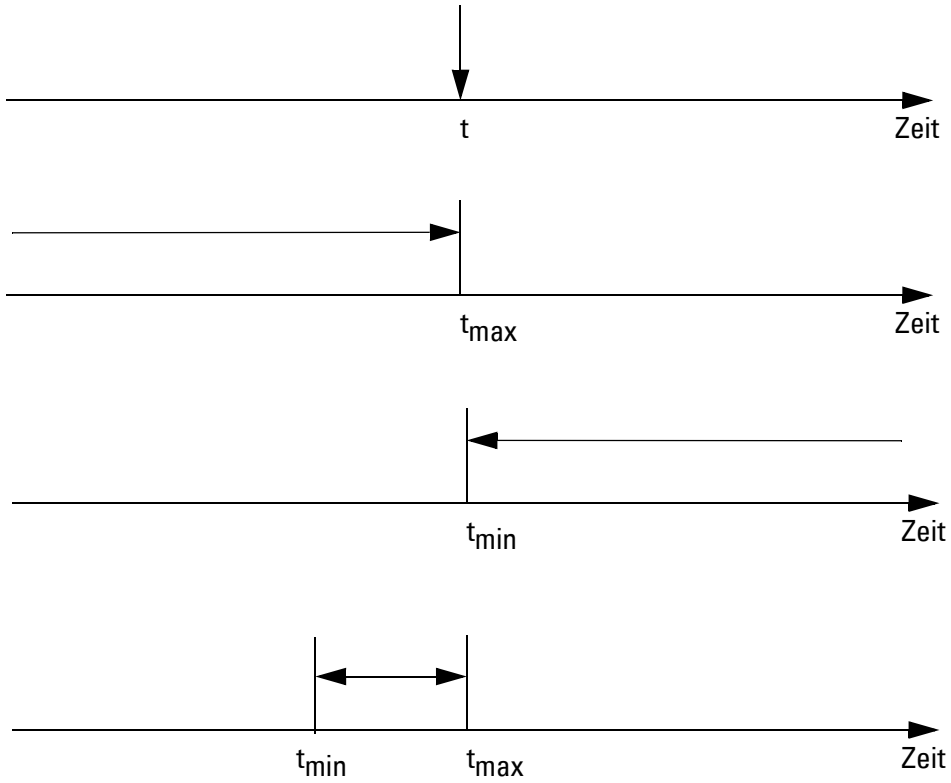
Erfordert indirekt auch die rechtzeitige Abholung von Eingangsdaten.

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)



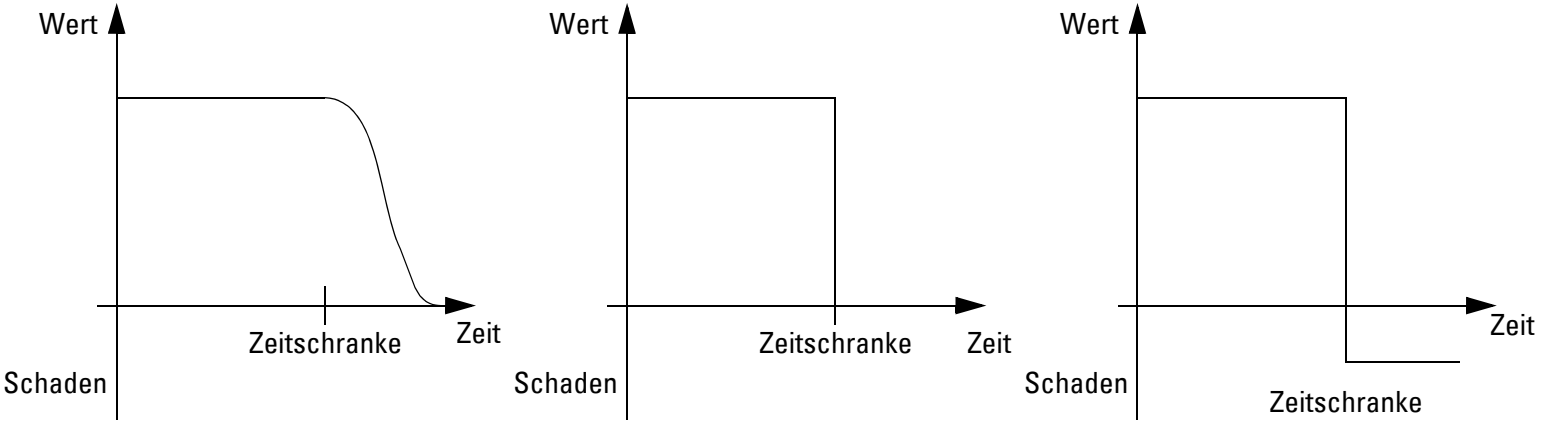
Unterteilung der *Zeitbedingungen* in

- *periodische* Zeitbedingungen
- *aperiodische* Zeitbedingungen

Weitere Unterteilung der *Zeitbedingungen* in

- *absolute* Zeitbedingungen
- *relative* Zeitbedingungen

Definitionen von harten, festen und weichen Echtzeitbedingungen:



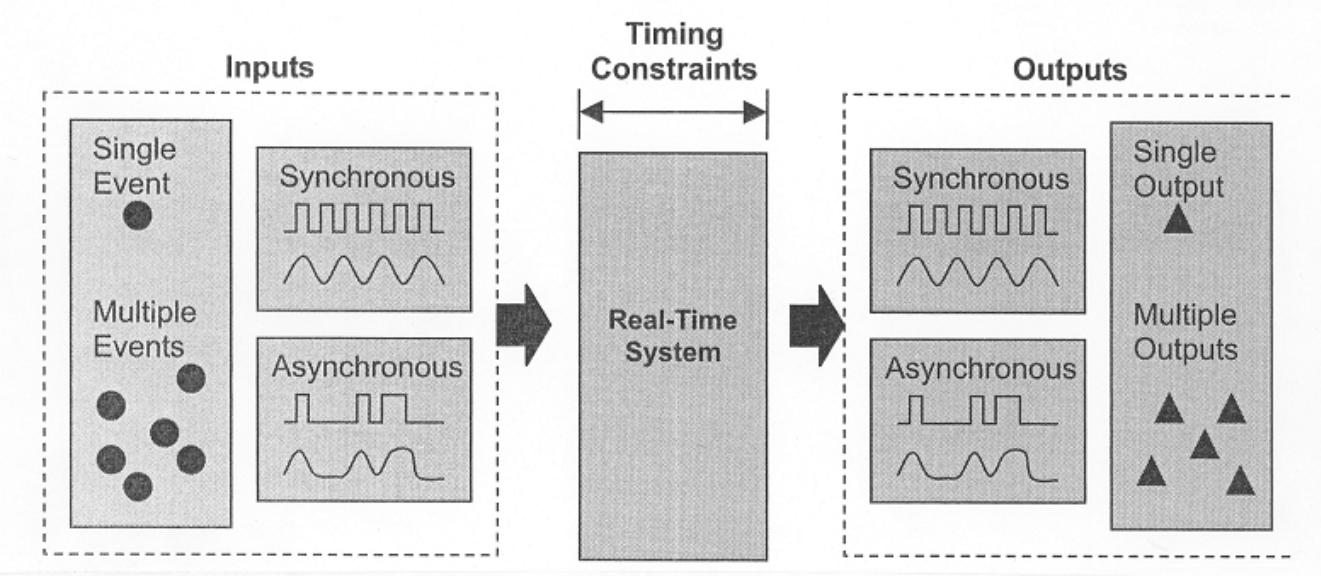
weich *fest* *hart*

Drei verschiedene Ansätze für Kriterien:

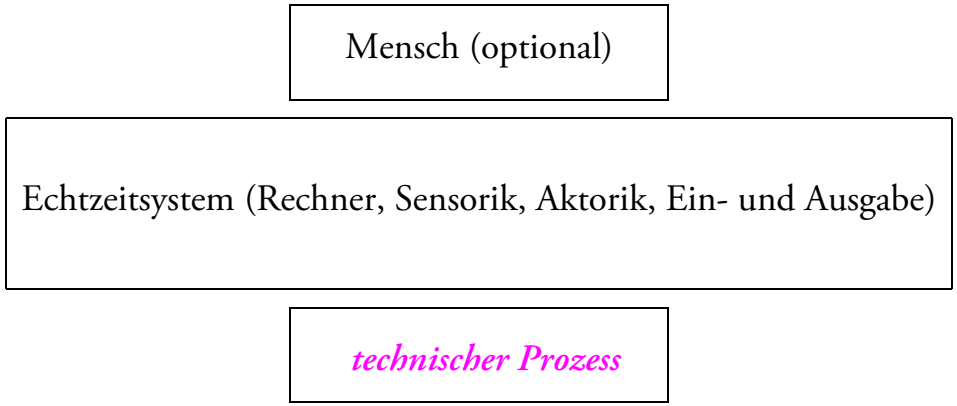
- nach Kritikalität (entsteht Schaden oder nicht)*; Beispiel: Airbag, Videoprozessor*
- nach Nützlichkeit (wie nützlich sind zu späte Ergebnisse)*; Problem hier: die Nützlichkeitsfunktion*
- nach zulässiger Häufigkeit von Zuspätkommen*; z.B. muss in 99,999% aller Fälle Zeitschranke einhalten*

1.8 Modell eines Echtzeitsystems

Ein einfaches Modell:



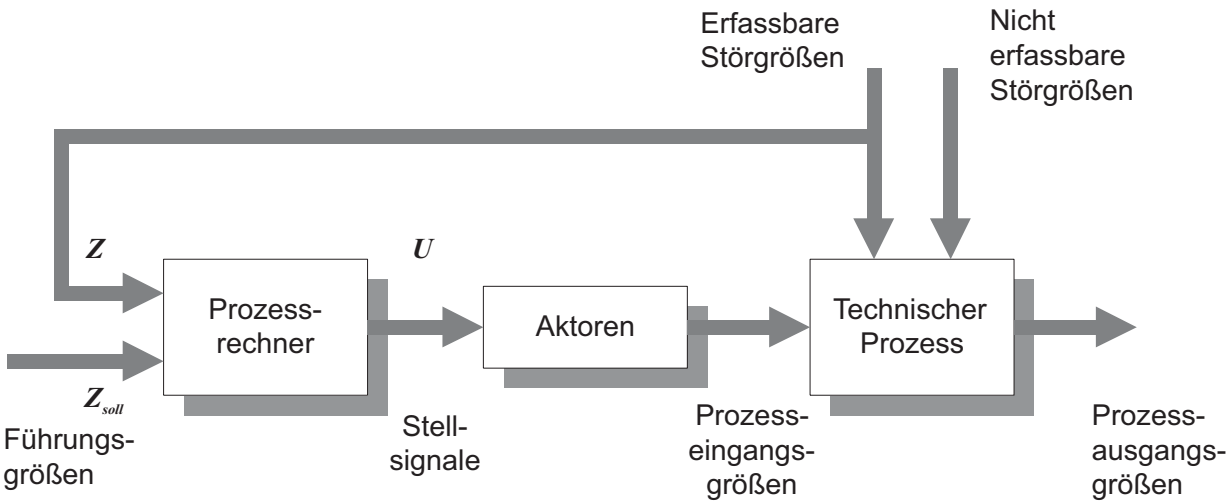
Andere Sicht:



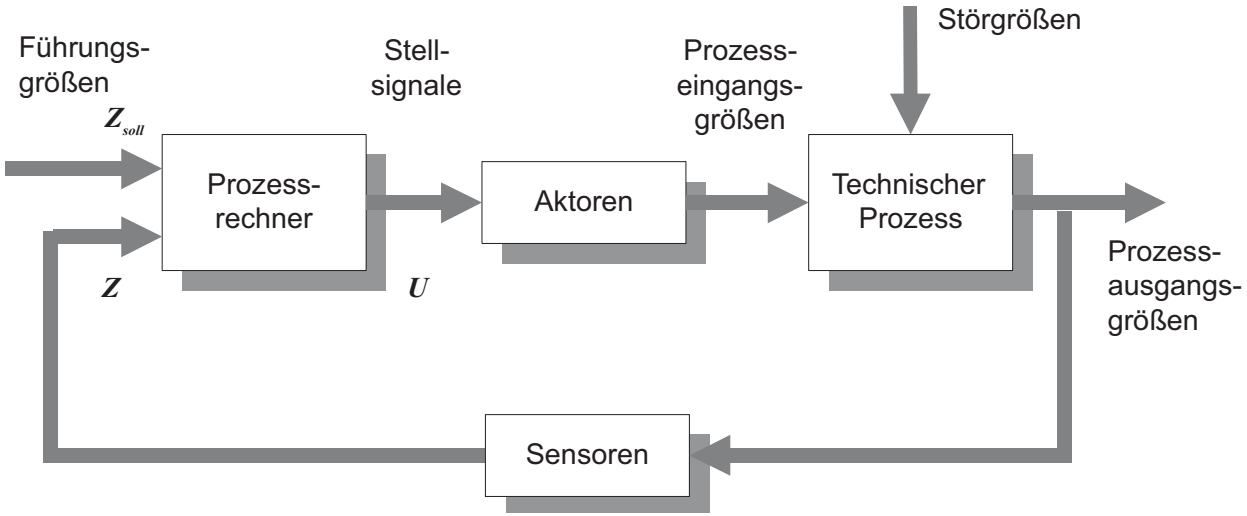
Echtzeitsysteme werden benutzt, um technische Prozesse zu

- *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.

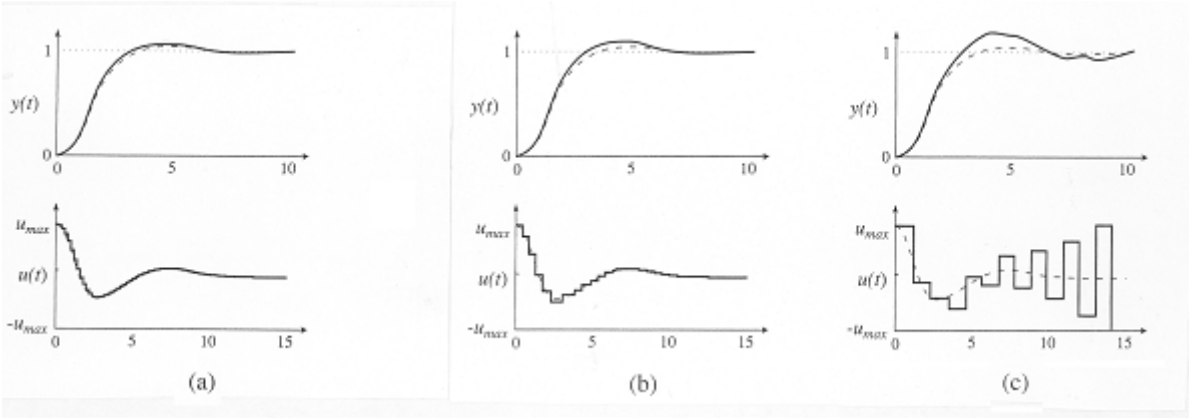
Prinzip der *Steuerung*:



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

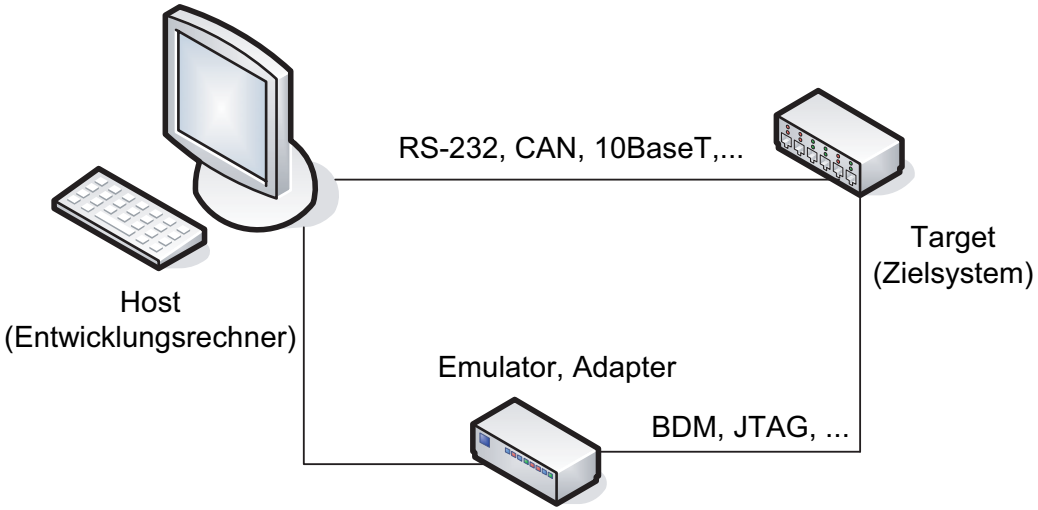


Abbildung 2-1: Entwicklungs- und Zielsystem

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

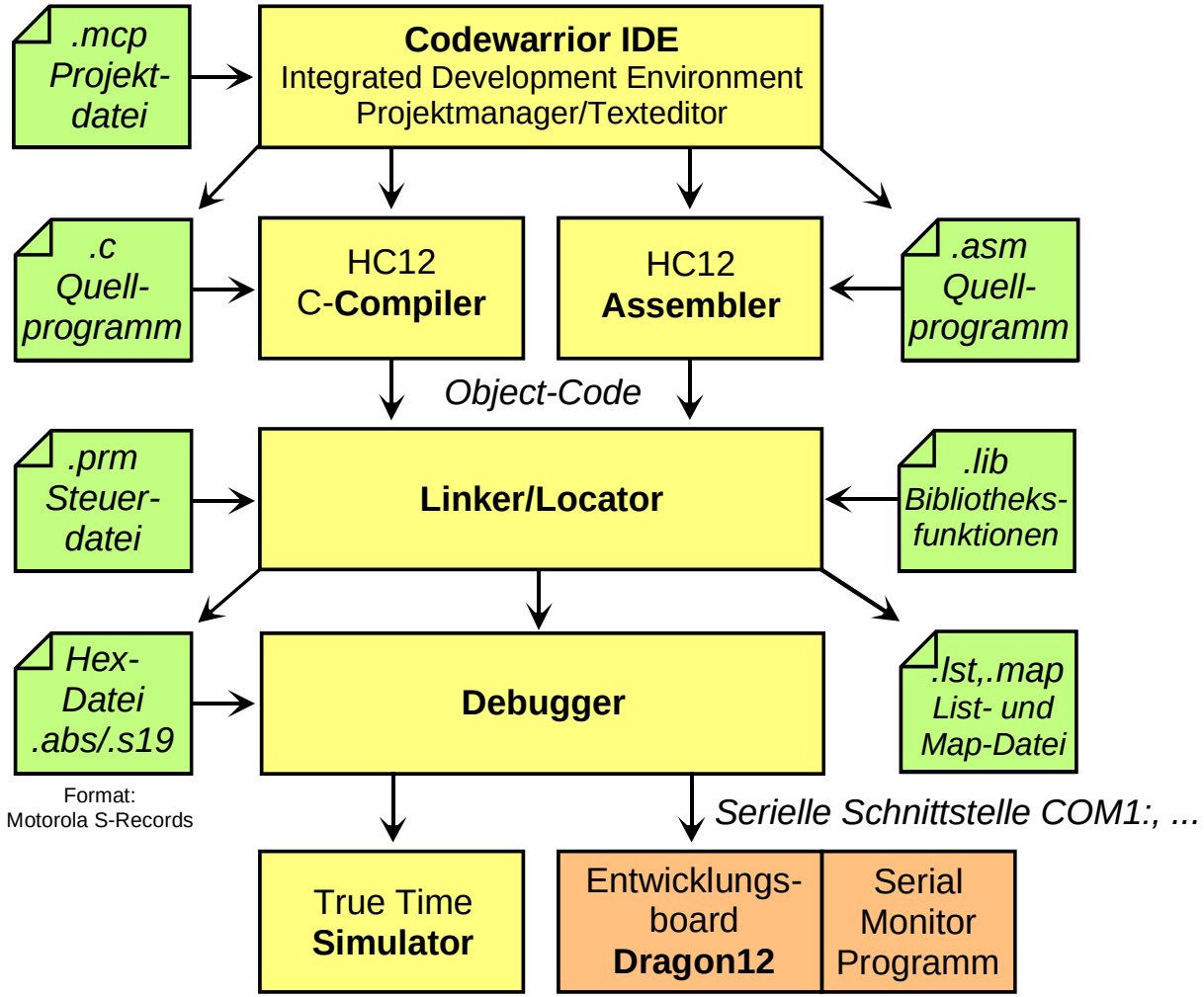
Standard-Entwicklungsrechner stellen heute USB-Anschlüsse und Ethernet-Anschlüsse sowie mit Einschränkungen RS-232-Schnittstellen zur Verfügung. Um JTAG-, BDM-, CAN- oder Emulator-Schnittstellen zu benutzen, ist ***zusätzliche Adapter-Hardware*** erforderlich.

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-

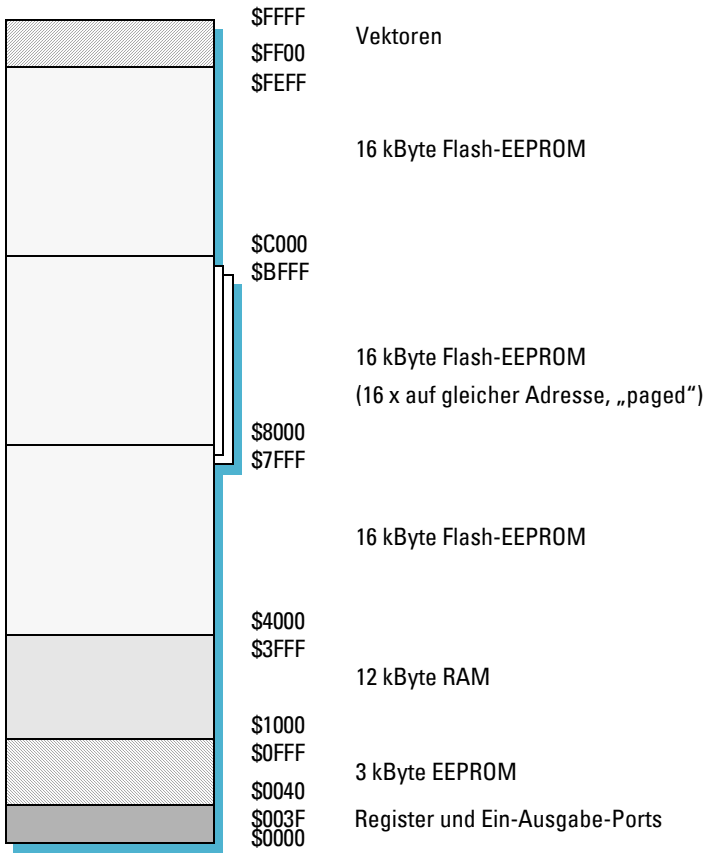
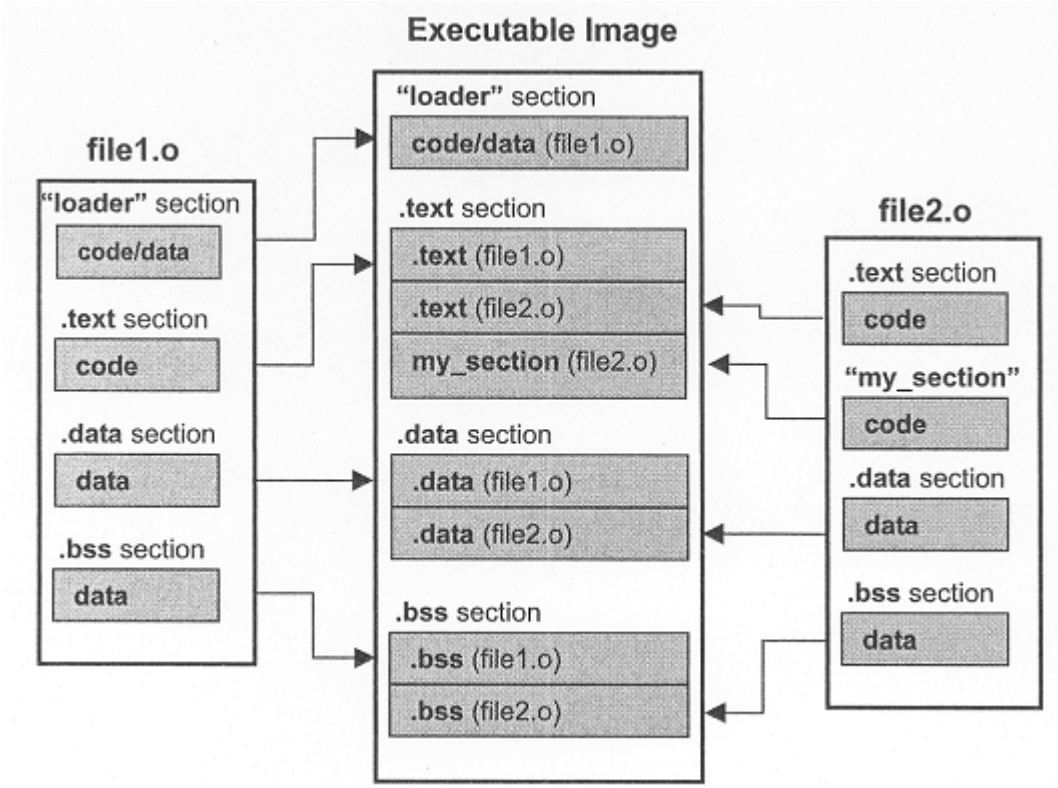


Abbildung 2-2: Memory-Map für den Freescale-Microcontroller 68HCS12

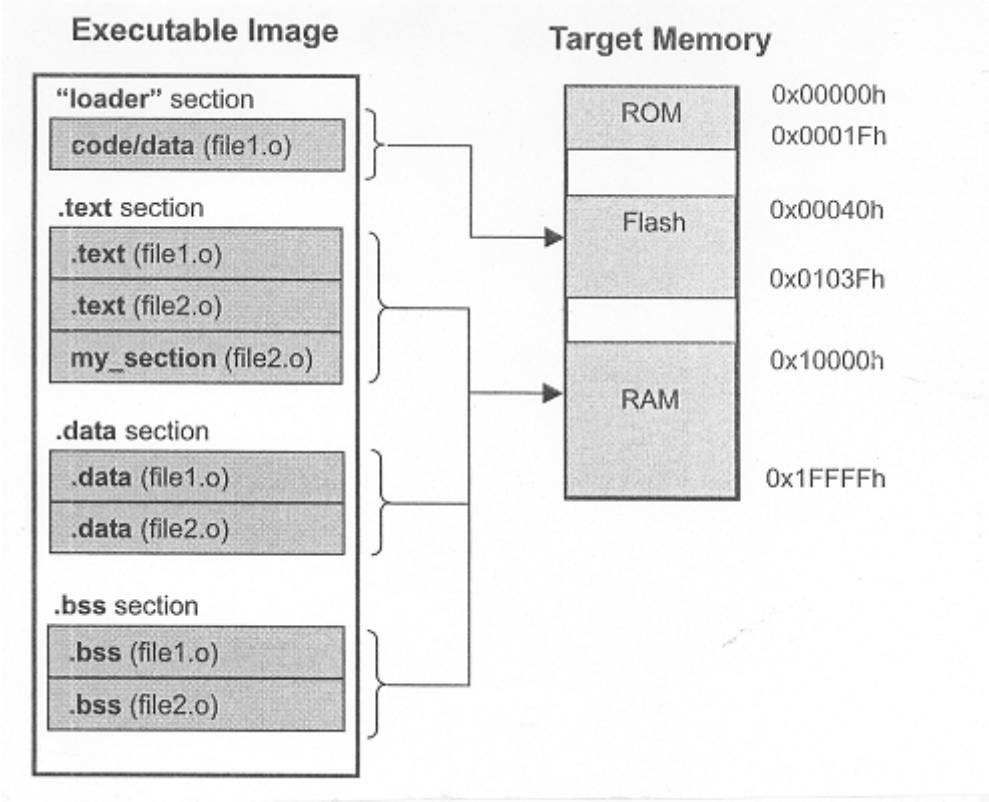
sung „Computerarchitektur 3“.

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.

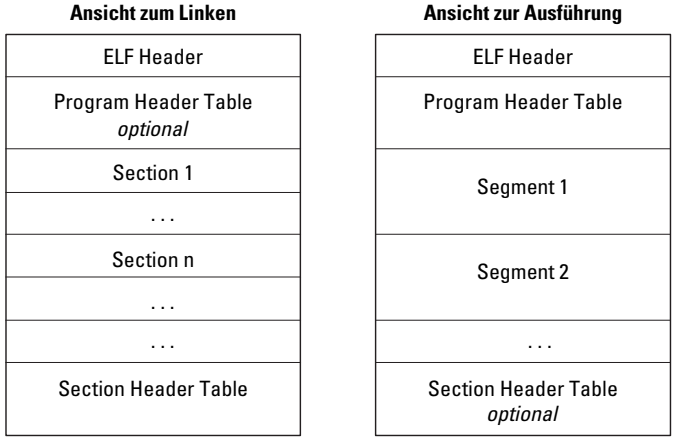


Bild 2.4: ELF-Dateistruktur

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:

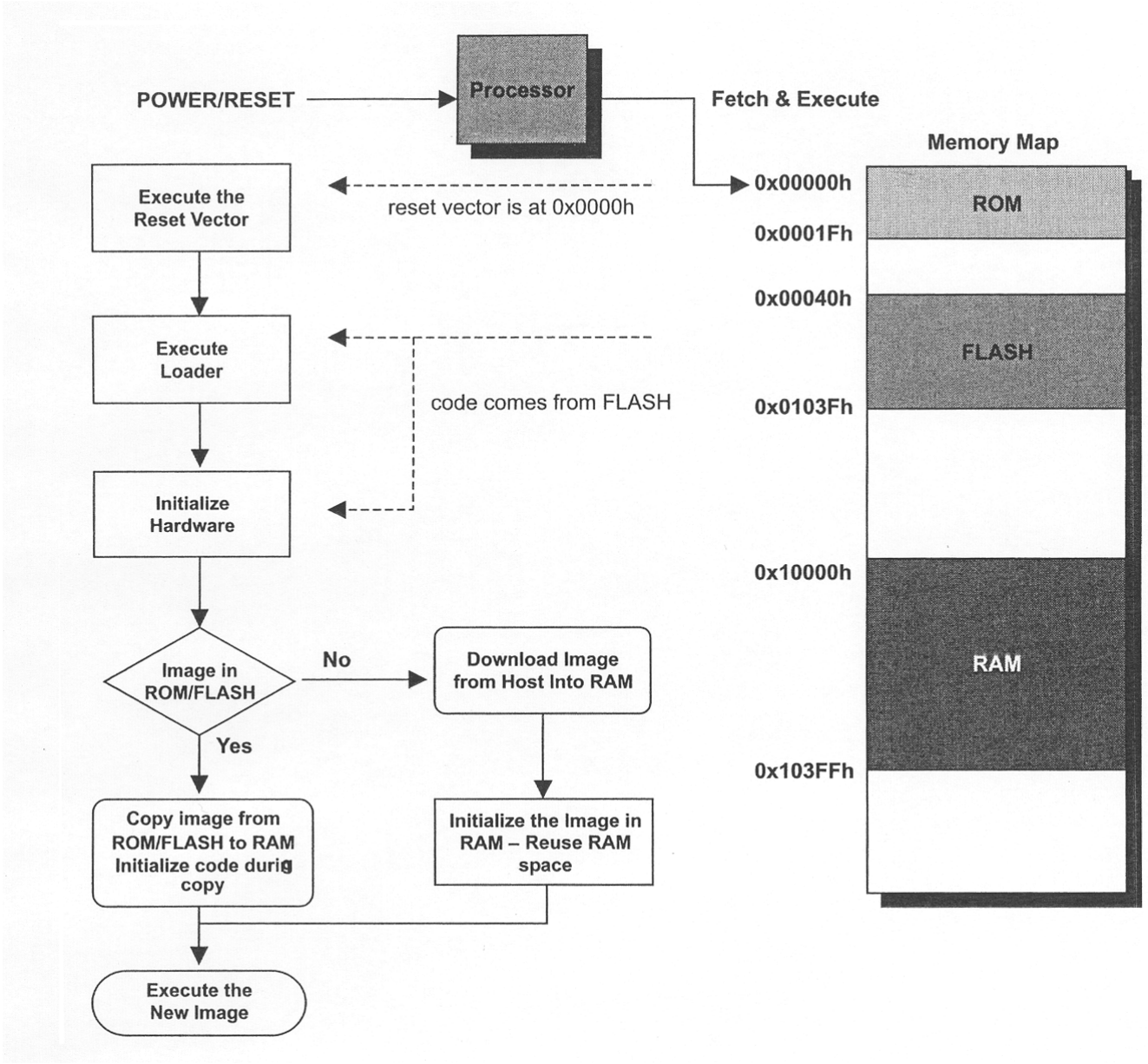
- Wenn Betriebssystemsystem vorhanden: Betriebssystem laden.

Schritt 3:

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

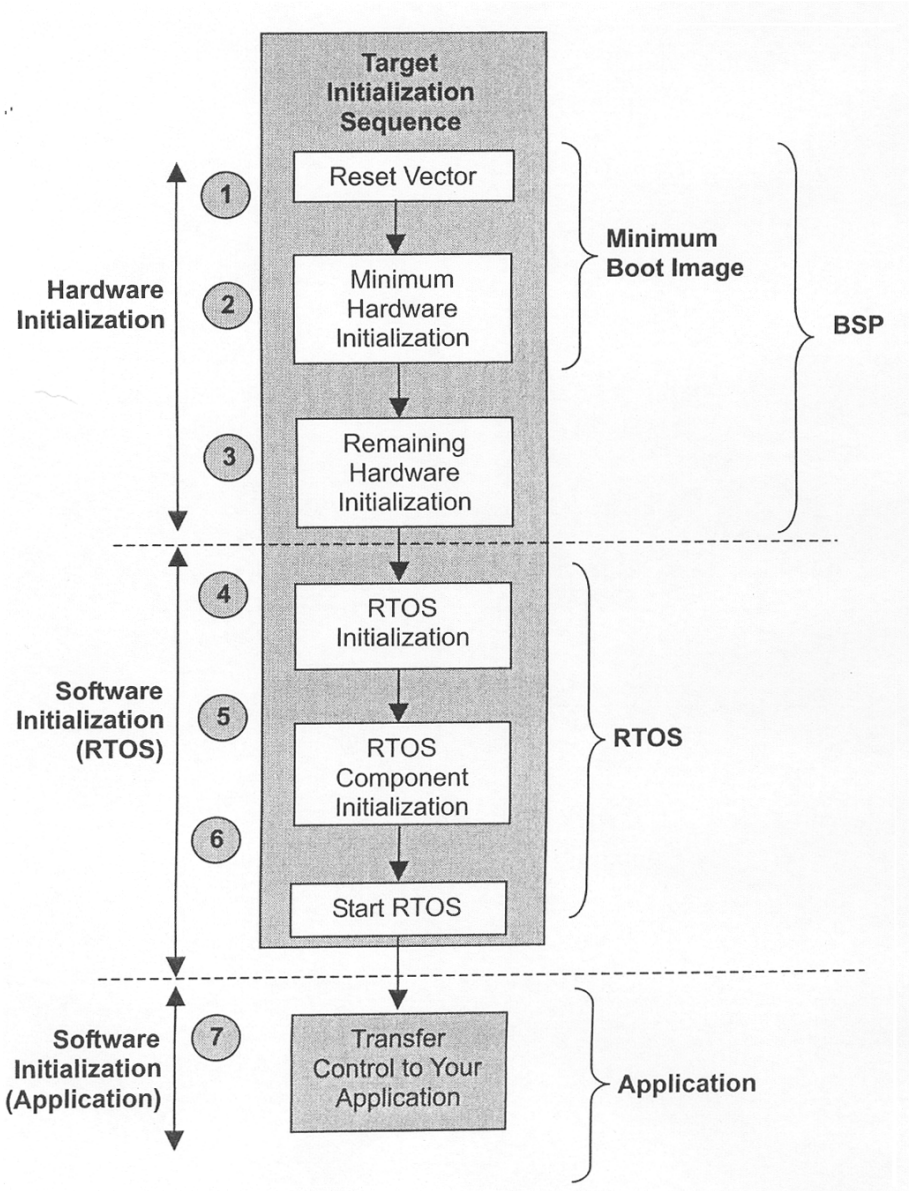
Schritt 4:

- 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

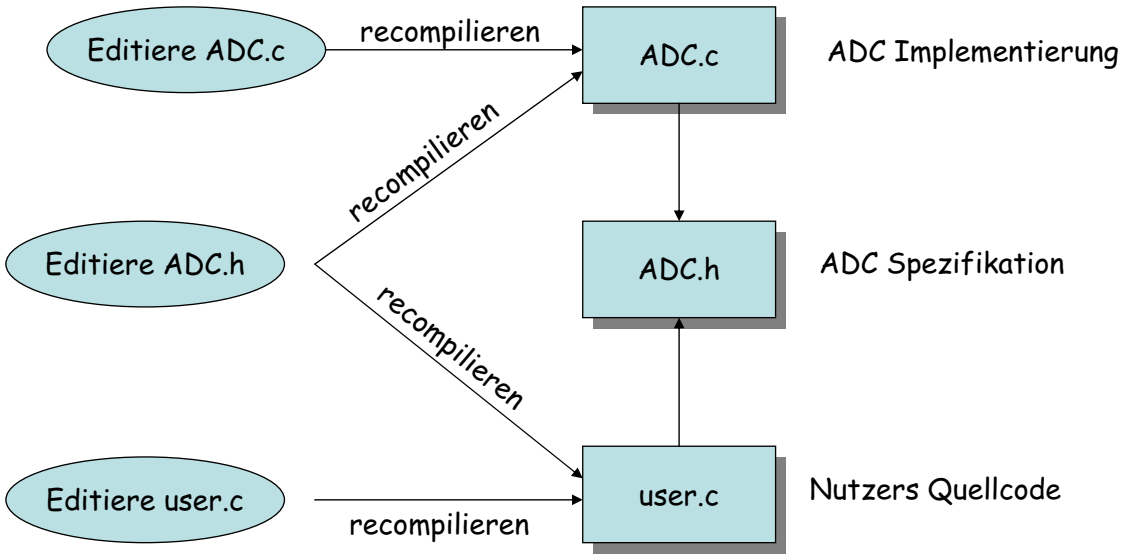
Weitere Regeln der Automotive-Industrie: MISRA (http://www.misra.org.uk). Dokument ist leider nicht frei verfügbar, muss man kaufen. Hier folgt meine persönliche Bestenliste.

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

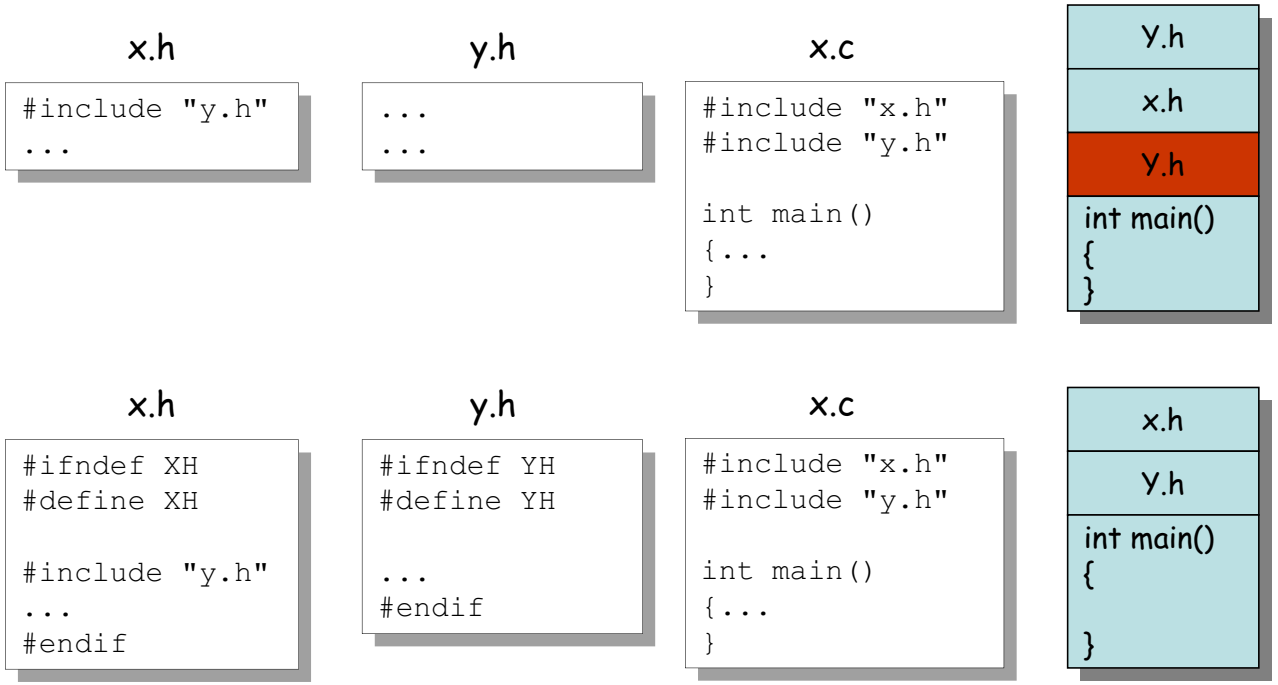
Siehe Darstellung. Wird bei einer IDE meist automatisch gehandhabt.



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

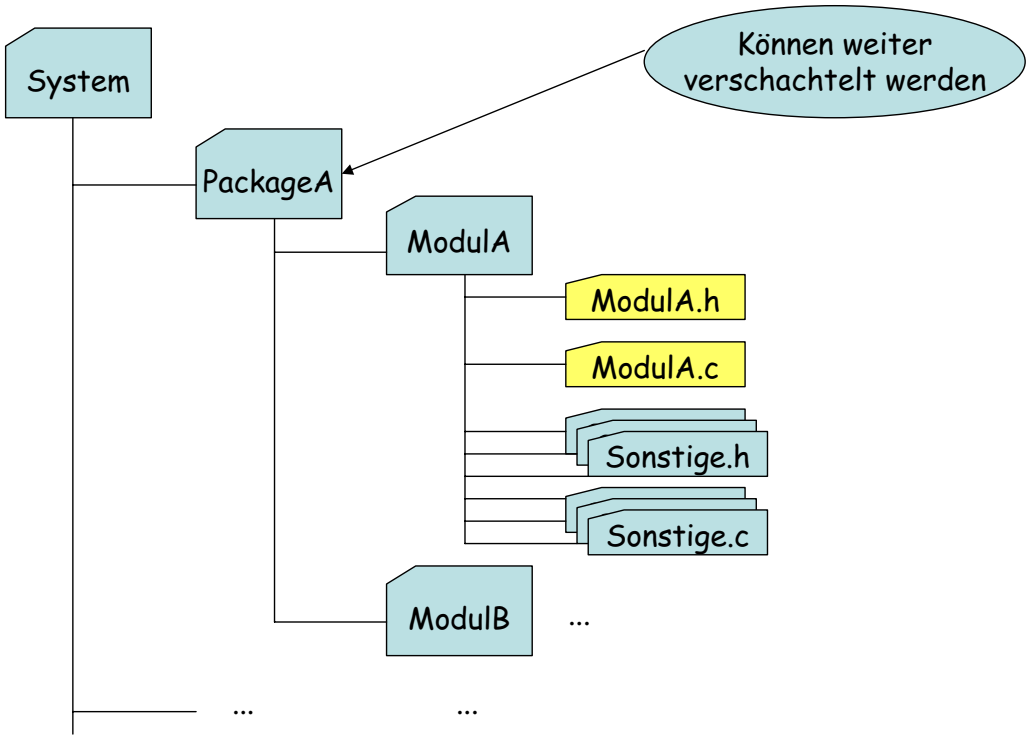
```
next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

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

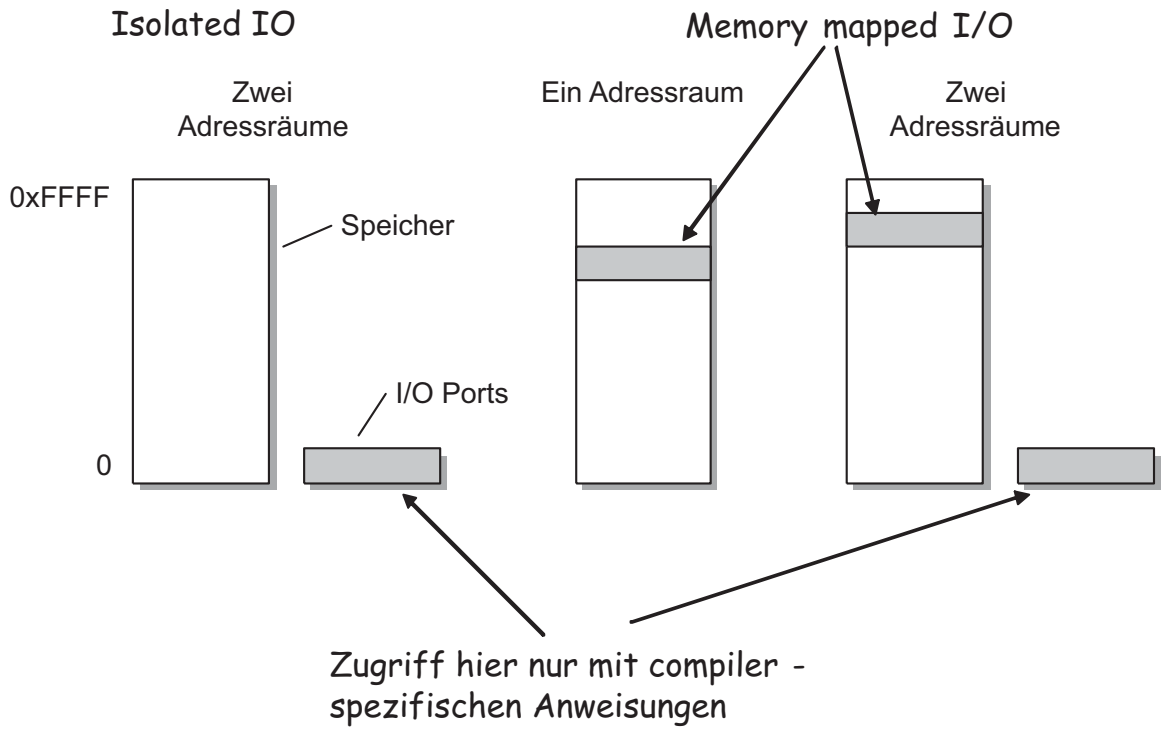
- *Pufferung* von Ein- und Ausgabedaten
- *Umsetzung* von Daten (z.B. seriell/parallel, analog/digital)
- Erzeugung von Steuer- und Handshake-Signalen
- Annahme und Erzeugung von *Unterbrechungsanforderungen*

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)



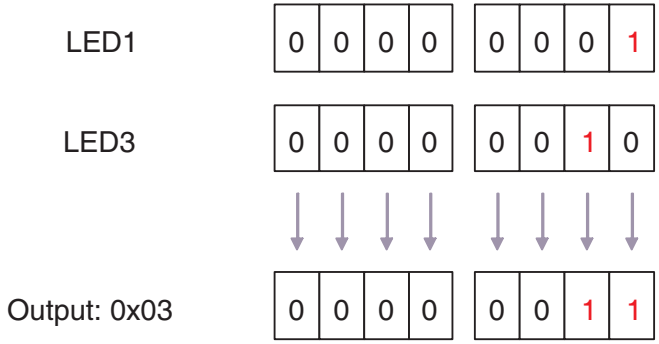
Beispiel für isolierte IO: Industrie-PC im Labor
Beispiel für Speicher-Ein-/Ausgabe: 68HCS12-Rechnersystem
Klassifizierung von Schnittstellen in

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

3.1.1 Bit-Manipulationen in C

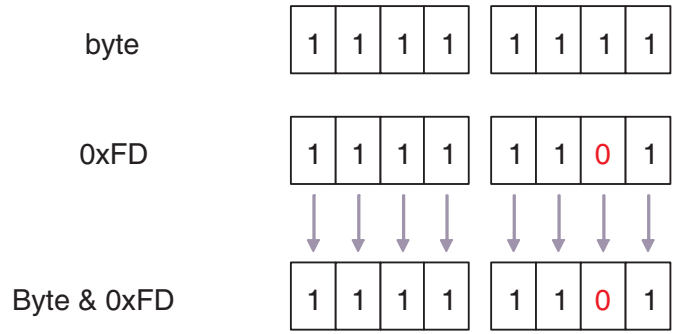
Bits setzen (die LED1, LED3 bezeichnet man als „*Maske*“):

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



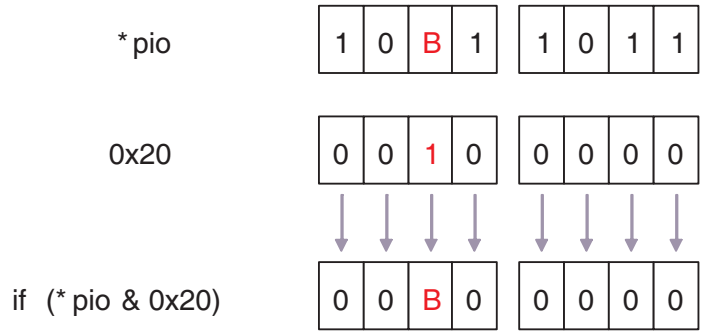
Bits löschen (die 0xFD nennt man „*Maske*“):

```
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
```



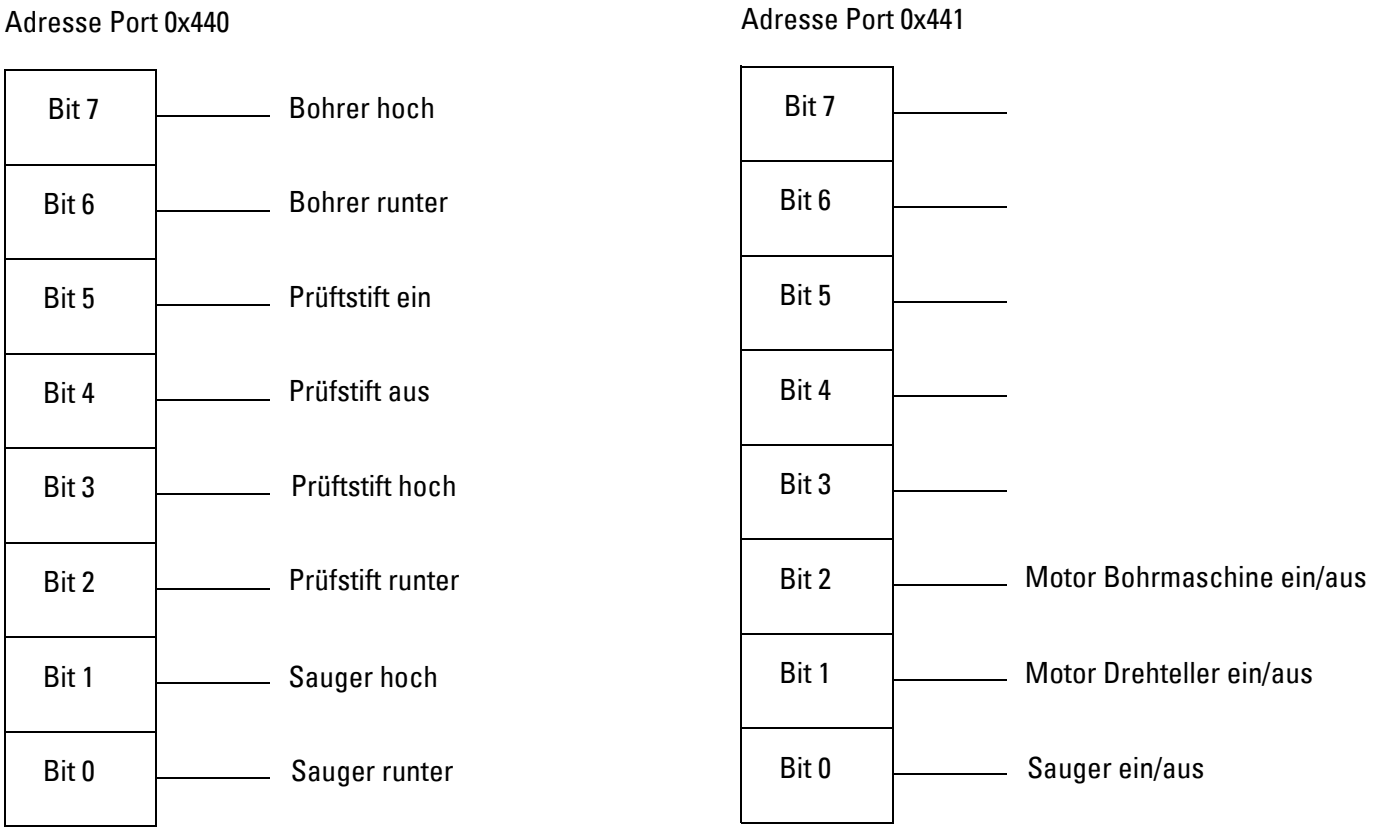
3.2 Beispiel: Parallele Ein- und Ausgabe auf SICOMP-Rechner

Der im Labor verwendete SICOMP-Rechner ist ein Industrie-PC, basierend auf der Intel-Architektur.

Peripheriegeräte sind an Parallelports angeschlossen: Bohrmaschinen-Motor, Drehteller-Motor, Pneumatik-ventile für Greifer, Prüfstift und Bohrmaschine.

Ansteuerung nur über spezielle Funktion möglich, da Parallelports nicht im Speicherbereich liegen.

Der Parallelports liegen an Adresse 0x440 bis 0x443. Belegung für Ausgänge:



```
1   ...
2   /*-------------------------------------------------------------*/
3   /* Digitale Ein-/Ausgabe 24V  AMS-P218                        */
4   /*-------------------------------------------------------------*/
5   #define P218_BASE       0x0440  /* Basisadresse               */
6   #define P218_CHN0       0x0440  /* Kanalgruppe 0 (Ausgabe)    */
7   #define P218_CHN1       0x0441  /* Kanalgruppe 1 (Ausgabe)    */
8   unsigned char portWrite(unsigned int io_adresse,
9                           unsigned char ausgabeWert,
10                          unsigned char maske) {
11    ...
12    unsigned char aus, x;
13    aus = ausgabeWert & maske ;
14    x  = inbyte (io_adresse); /* Spezielle Funktion nötig, kann nur in */
15                              /* erstellt werden Assembler */
16    x  = ( x & ~maske ) | aus ;
17    outbyte (io_adresse , x ) ;/* Spezielle Funktion nötig, kann nur in */
18                              /* erstellt werden Assembler */
19    ret = inbyte (io_adresse) ;
20    return (ret);
21  }
22
```

3.3 Beispiel: Zeitgeber (Timer) auf Dragon12-Board

Echtzeitanwendungen benötigen oft eine Möglichkeit, Zeit oder Zeitdifferenzen zu messen.

Dafür besitzen Rechner *Zeitgeber*.

Zeitgeber (Timer) bestehen aus Hardware-Digitalzählern mit Steuerlogik.

Zeitgeber laufen unabhängig von der Anwendungssoftware, können von dieser aber gesteuert und beobachtet werden.

Zeitgeber können über *Unterbrechungssignale* die Anwendungssoftware *unterbrechen.*

Unser Mikrocontroller besitzt sehr umfangreiches Zeitgebermodul. Funktionen:

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

Zentraler Bestandteil des Zeitgebers: durch Oszillator getriebenener 16-Bit-Zähler.

Kann zur Messung relativer Zeiten benutzt werden (Lesen von Software aus).

Darüber hinaus: 8 konfigurierbare *Zeitgeber-Kanäle*

- Im *„Input Capture"-Modus* wird Wert des 16-Bit-Zählers in Register übernommen, wenn am zugeordneten Eingangspin ein Signalübergang detektiert worden ist. Gut zur Frequenz- und Periodendauermessung.
- Im *„Output Compare"-Modus* wird zugeordneter Ausgangspin auf bestimmten Pegel gesetzt oder verändert, sobald 16-Bit-Zähler im Vergleichsregister voreingestellten Wert erreicht hat. Gut zur Erzeugung zeitlich genau definierter Ausgangssignale z.B. für die Steuerung von Servomotoren.
- 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 wird von allen Kanälen benutzt.
- Zähler ist über das Register TCNT lesbar.
- Der Zähler wird nicht direkt vom Bustakt getaktet, sondern über einen *Vorteiler (Prescaler).*
- Prescaler ist nach Reset auf Teilfaktor von 1 eingestellt.
- Über dass 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.

Beispiel: Vorgehensweise zur *Messung* einer *Zeitdifferenz*:

1. Aktuellen Wert von TCNT als Anfangszeit speichern.
2. Zum Messzeitpunkt TCNT wieder lesen, und von Anfangswert abziehen:

```
1       movw   TCNT, startZeit; startZeit ist eine Wort-Variable
2       ; hier kann jetzt etwas passieren, von dem wir die Zeit messen wollen
3       ldd    TCNT      ; hier laden wir die neue Zeit
4       subd   startZeit ; in D steht jetzt die Zeitdifferenz
```

- Absoluter Wert von TCNT nicht relevant
- Zählerüberlauf spielt keine Rolle. Einzige Einschränkung: maximal *65535 Zähleinheiten messbar.*

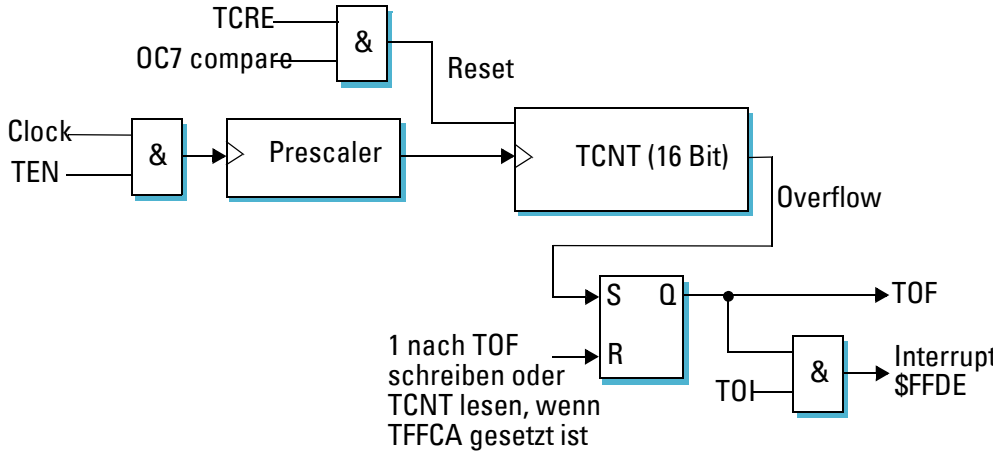


Abbildung 3-1: Konfiguration des freilaufenden Zählers

3.3.2 TOF-Bit (Timer Overflow Flag)

- Immer wenn der Zähler überläuft, wird das *TOF-Status-Bit gesetzt.*
- Rücksetzen des TOF-Bits durch Schreiben einer 1 nach TOF.
- Wenn TFFCA-Steuer-Bit gesetzt, wird TOF durch Lesen des 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

3.3.3 TOI-Bit (Timer Overflow Interrupt)

- TOI-Bit gesetzt: Zählerüberlauf führt zu Interrupt.
- Anwendungsbeispiel: *Hardwarezähler* um *Softwarezähler* von praktisch beliebiger Länge erweitern.
- Konkretes Beispiel für die Erweiterung auf 32 Bit:

```
1   timerInterruptServiceRoutine:
2      ldd    timerExtension; liegt irgendwo im Speicher (ds.w 1)
3      addd   #1
4      std    timerExtension
5      movb #$80,TFLG2; Interrupt-Flag zurücksetzen
6      rti
```

- Mögliches Problem beim Lesen des Zählers, da sich Wert während des *Lesens ändern kann.*
- Passiert selten, aber es kann passieren:

```
1      movw   timerExtension, startZeit
2      ; schlecht, wenn hier ein TOI kommt...
3      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 (Zeile 1) die Interrupt-Service-Routine (ISR) angesprungen wird.
- ISR inkrementiert `timerExtension`, während TCNT wieder bei 0 beginnt.
- Wenn Rechner aus ISR zurückkehrt und TCNT gelesen hat, steht in `startZeit` und `startZeit+2` ein inkonsistenter Wert (wir sind in der Zeit zurückgefallen).

3.3.4 Beispiel: 10 ms Ticker für ersten Laborversuch

Initialisierung

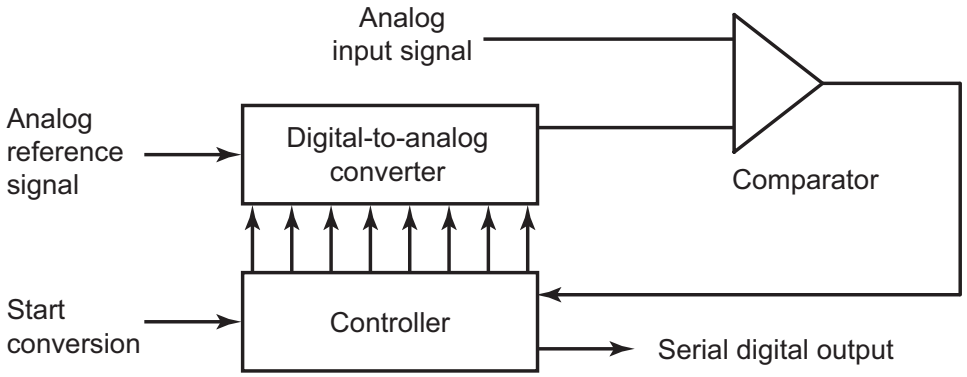
```
1   /*
2    * Initialize the timer 4 hardware and interrupt
3    *
4    */
5   void initTicker(void) {
6
7     tickerFunctionPointer = 0;
8     ticks = 0;
9     /* Timer master ON switch  */
10     TSCR1 = TIMER_ON;
11
12     /* Set channel 4 in "output compare" mode  */
13     TIOS = TIOS | TIMER_CH4; /* bit 4 corresponds to channel 4 */
14
15     /* Enable channel 4 interrupt; bit 4 corresponds to channel 4 */
16    TIE = TIE | TIMER_CH4;
17
18    /* Set timer prescaler (bus clock : prescale factor)        */
19    /* In our case: divide by 2^7 = 128. This gives a timer     */
20    /* driver frequency of 187500 Hz or 5.3333 us time interval */
21    TSCR2 = (TSCR2 & 0xf8) | 0x07;
22
23    /* switch timer on */
24    TCTL1 = (TCTL1 & ~TCTL1_CH4) | TCTL1_CH4_DISCONNECT;
25  }
```

Unterbrechungsroutine

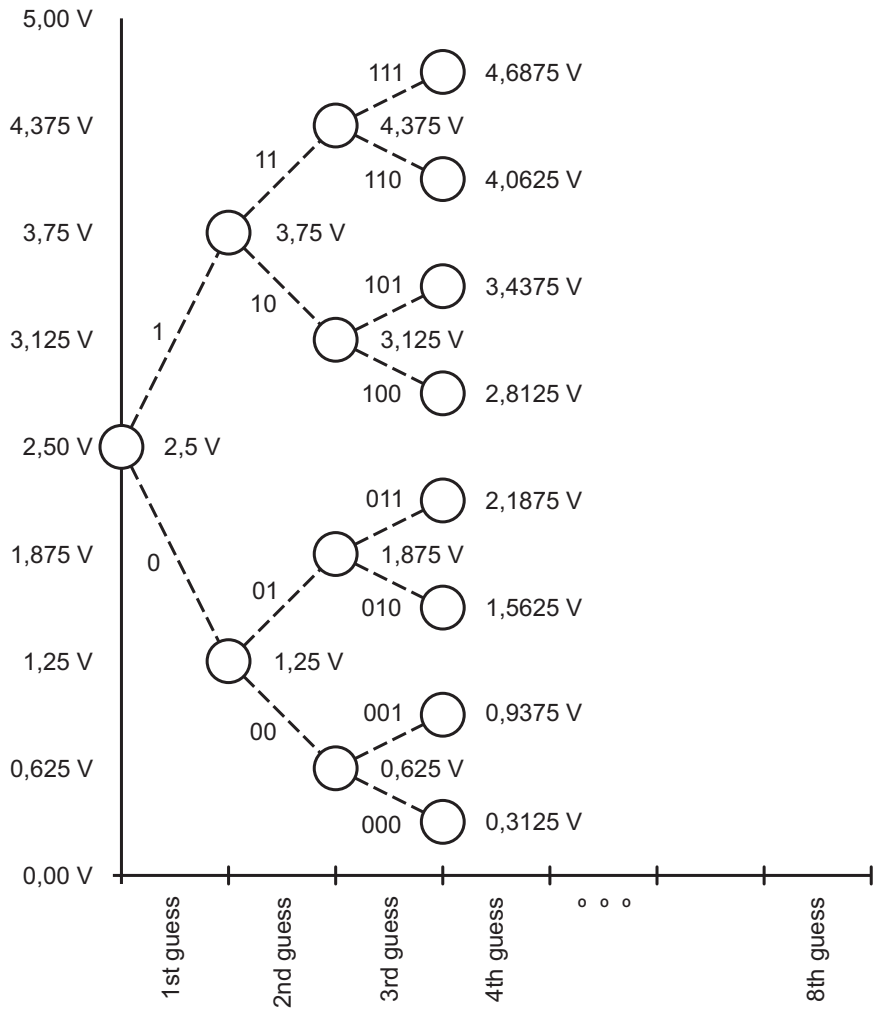
```
1   /*
2    * Interrupt service routine for timer channel 4
3    *
4    */
5   interrupt void isr_tc4(void) {
6
7      /* increment timer count register (16 bits) */
8      /* current count + increment = new count    */
9      TC4 = TC4 + TENMS;
10     // ++ticks;
11     // if (ticks > 100) {
12     //     ticks = 0;
13     //   PORTB = ~PORTB & 0x01;
14     // }
15     if ((void *) tickerFunctionPointer > NULL) {
16       tickerFunctionPointer();
17     }
18     /* clear the interrupt: write a 1 to bit 4 */
19    TFLG1 = TIMER_CH4;
20  }
```

3.4 Beispiel: Analog-Digitalwandler auf Dragon12-Board

- Viele Anwendungen erfordern eine Anbindung des Rechners an analoge Sensoren, z.B. Beschleunigungsaufnehmer, Temperaturfühler oder Drucksensoren.
- Damit Rechner Signale verarbeiten kann, müssen sie zuerst *digitalisiert* werden.
- Freescale-Controller besitzt zwei unabhängige 10 Bit A/D-Konverter
- Wandler arbeiten nach dem Verfahren der *sukzessiven Approximation*



- Referenzspannung werden durch VRH und VRL vorgegeben (5 und 0 V auf unserem Board)
- *Auflösung* ist damit (5.0-0.0)/1024 = 4,88mV
- *Umsetzzeit* ist *konstant*, da Anzahl der Vergleiche durch Auflösung festgelegt ist
- Interne Logik probiert Bit für Bit, beginnend mit wichtigstem Bit, ob durch D/A-Wandler erzeugter Wert größer oder kleiner als Eingangssignal ist



- 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 (10 Bit in 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.

3.4.1 Initialisierung

- 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 ATD0CTL2 bis ATD0CTL4 konfiguriert werden.
- Abschluss Konfigurationsphase durch und Start einer Wandlung durch Schreiben in das Steuerungsregister *ATD0CTL5*
- Das ADPU-Bit aktiviert das A/D-Wandlermodul (Voreinstellung: abgeschaltet, da relativ hoher Stromverbrauch)
- 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.
- Bits S8C bis S1C bestimmen die Anzahl der Wandlungen pro Befehl. Man darf 1 bis 8 Wandlungen einstellen; andere Werte bedeuten *8 Wandlungen.*

3.4.2 Wandlungsergebnisse

- 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 3-2: 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. Beispiel: Um bei 24 MHz Bustakt maximale A/D-Wandler-Takt zu erzielen, muss Vorteiler auf *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 3-3: 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 3-3 gezeigt aus.

3.4.3 Kanalselektion

- Jeder Analog-Digitalwandler kann mit einem von acht Eingängen verbunden werden. Dazu dienen die Bits CA bis CC im Register ATDxCTL5.
- Ist 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.

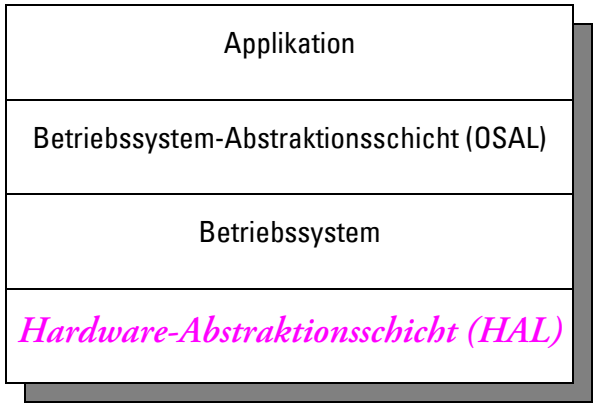
- Eine Wandlungssequenz wird ansonsten begonnen, indem man in das Register *ATDxCTL5 schreibt.*
- Sobald ein Ergebnis in ein Register ADRxx gespeichert worden ist, wird das entsprechende Flag CCFx gesetzt.
- Die CCx-Bits in Register ATDxSTAT0 zeigen an, wohin das nächste Ergebnis gespeichert werden wird.
- Das Bit AFFC in ATD0CTL2 bestimmt den „Fast Flag Clear All“-Modus des A/D-Wandlers.
- Steht AFFC auf 0, 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 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*.

3.6 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:

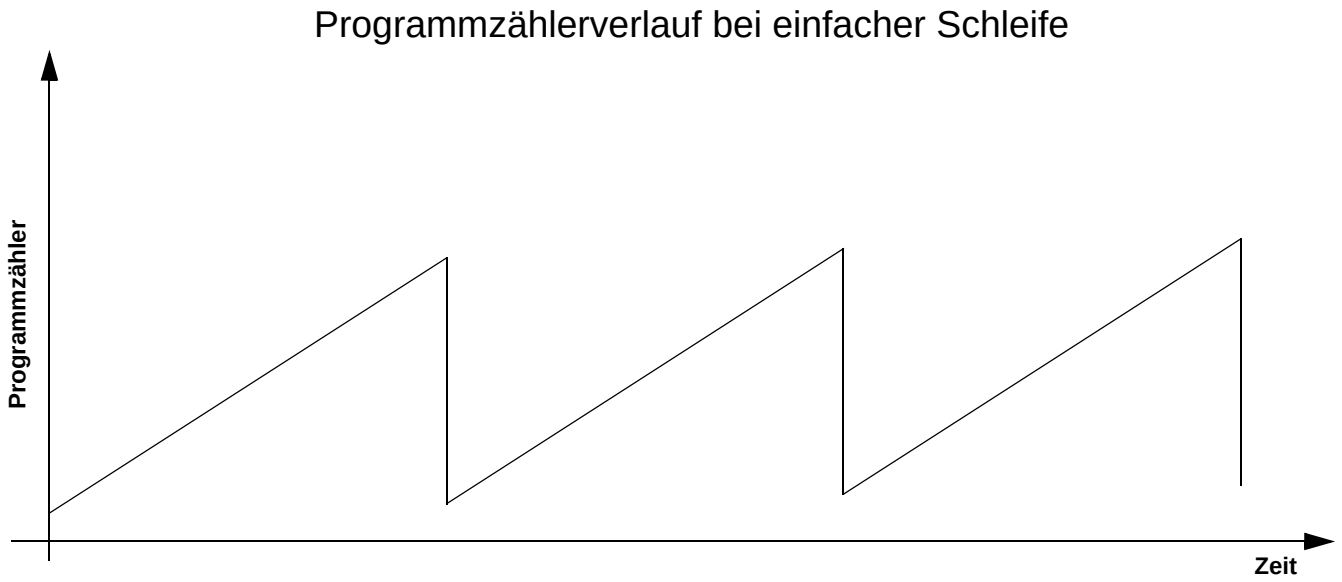
```
1   int main() {
2      /* do here what needs to be done once ... */
3      for (;;) {
4         /* do here what needs to be done all the time...*/
5      }
6   }
```

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.

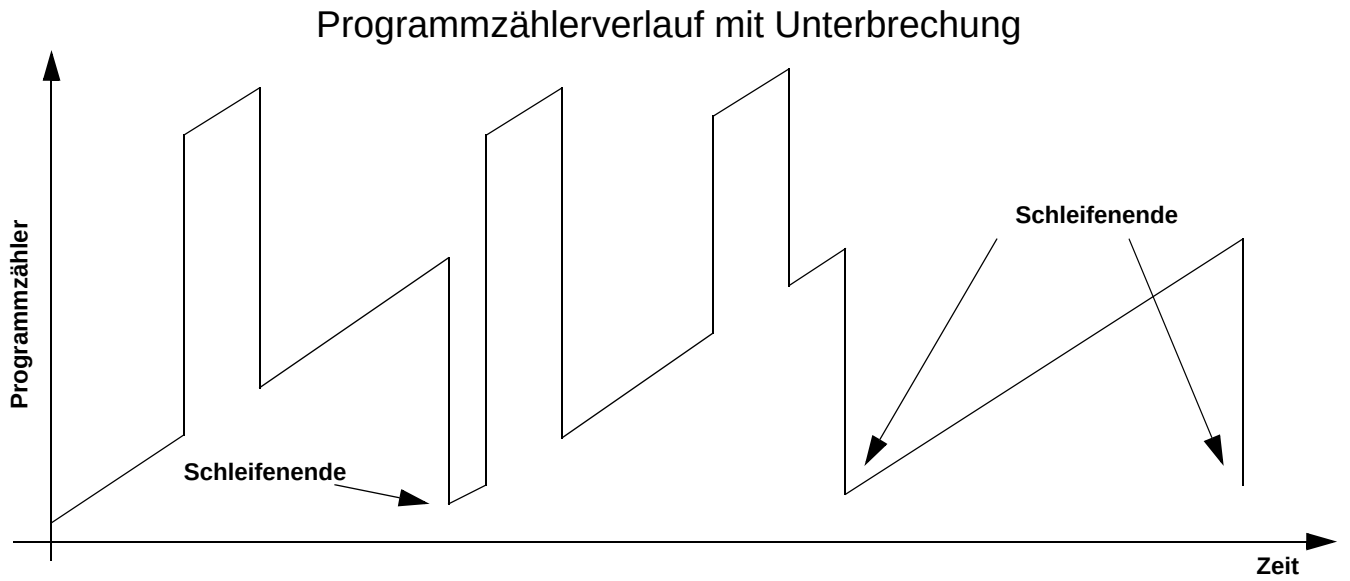
Wie sagt er es unserem Programm? Er unterbricht es kurz durch eine *Unterbrechungsroutine*!



- 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)

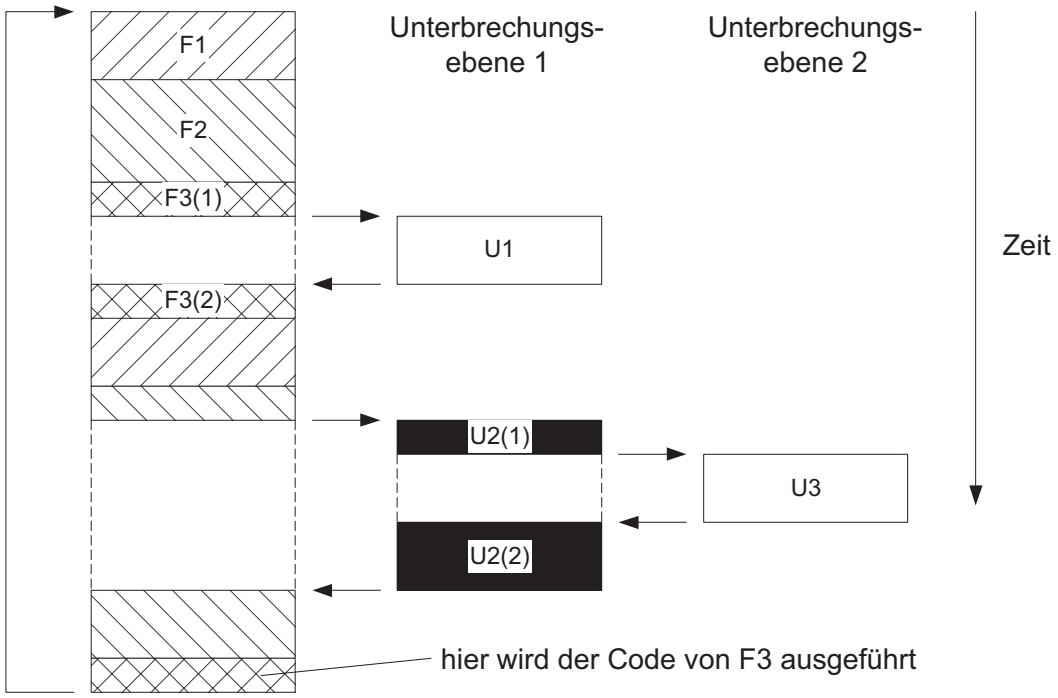
Die meisten Rechner bieten die Möglichkeit, aufgrund von *Hardwaresignalen* oder Ereignissen den Verlauf des Programmzählers zu ändern.



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:



3.7 Unterbrechungsbehandlung

Rechner arbeiten Programme 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 3-4: 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 3-4: 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*):

3.7.1 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 3-4).
- 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.

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

Interrupt-Serviceroutinen lassen sich in Assembler oder in C programmieren. Wir nehmen unser Blinklichtprogramm 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  }
```

3.8 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

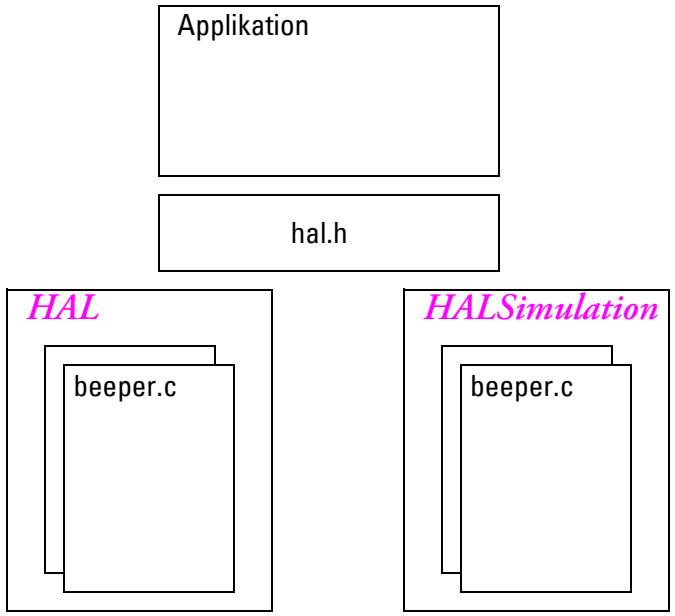
Trotzdem: wenn immer denkbar soll dieser Ansatz verwendet werden.

3.8.1 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.



3.8.2 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.

3.8.3 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.

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.

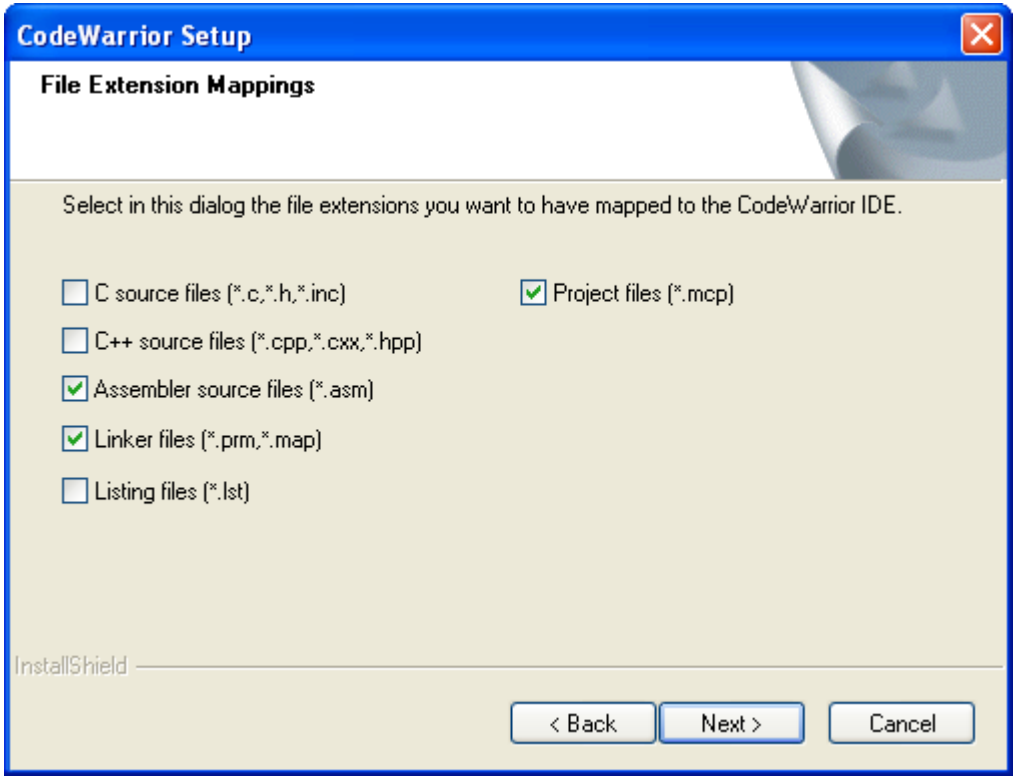


Abbildung A-1: Stellung des File Extension Mappings

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.

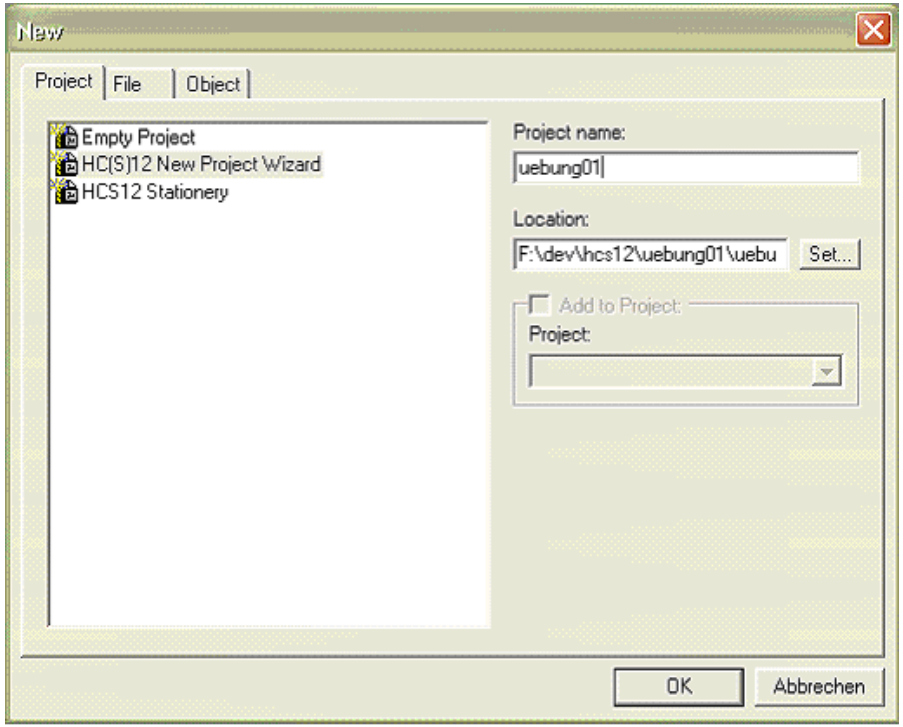
Das Board muss im EVB-Modus konfiguriert sein, damit es mit dem Debugger zusammen arbeitet. Das bedeutet, dass SW7 so eingestellt sein muss, das die linke LED (EVB) rechts unten in der Ecke leuchtet.

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.

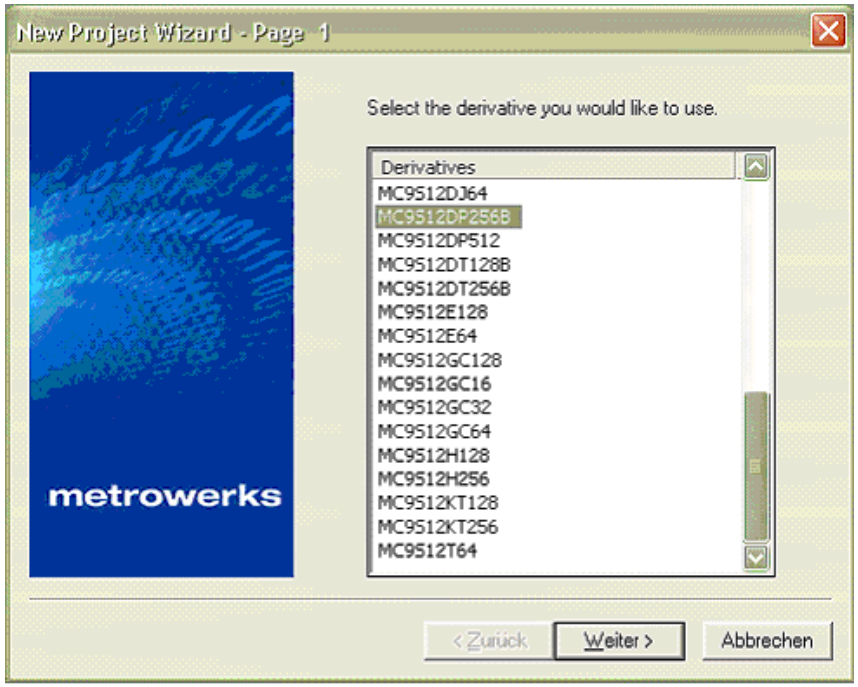
B.3 Anlegen eines neuen Projekts

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.

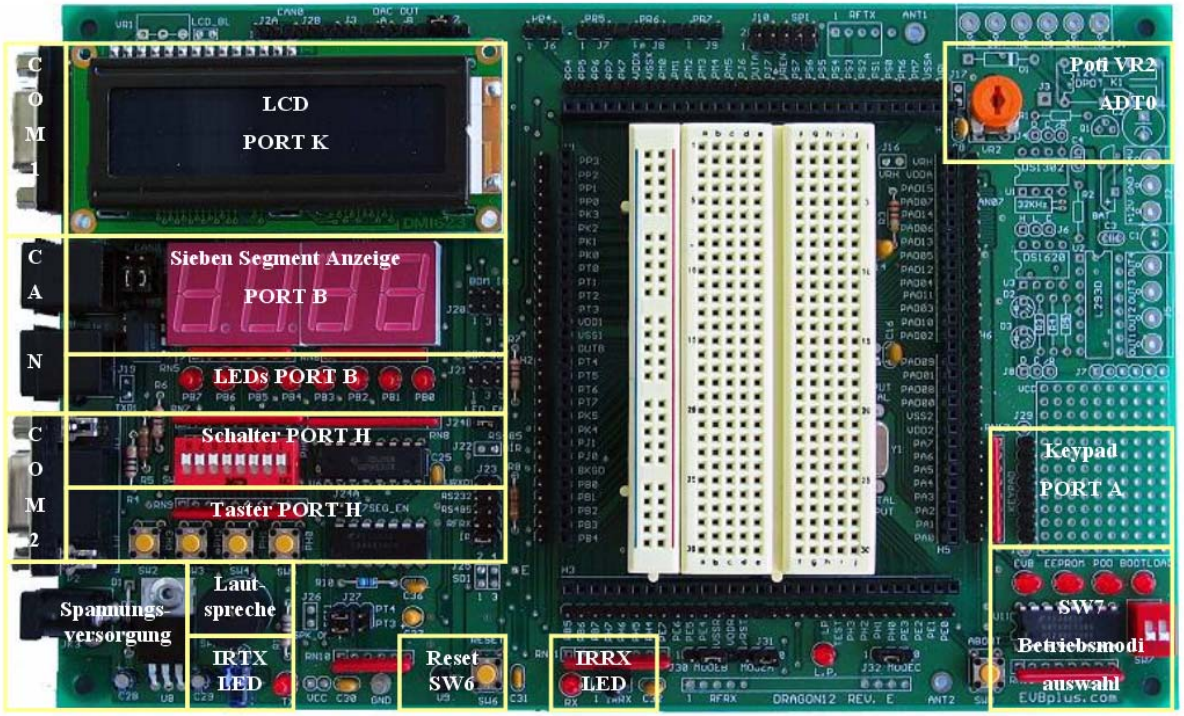


Abbildung B-1: Das im Labor verwendete Dragon12-Board von Wytec

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:

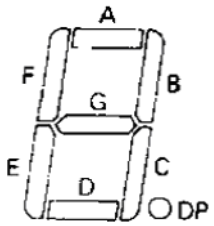
```
...
DDRJ_DDRJ1 = 1;  /* LED-Zeile deaktivieren */
PTJ_PTJ1    = 1;

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.

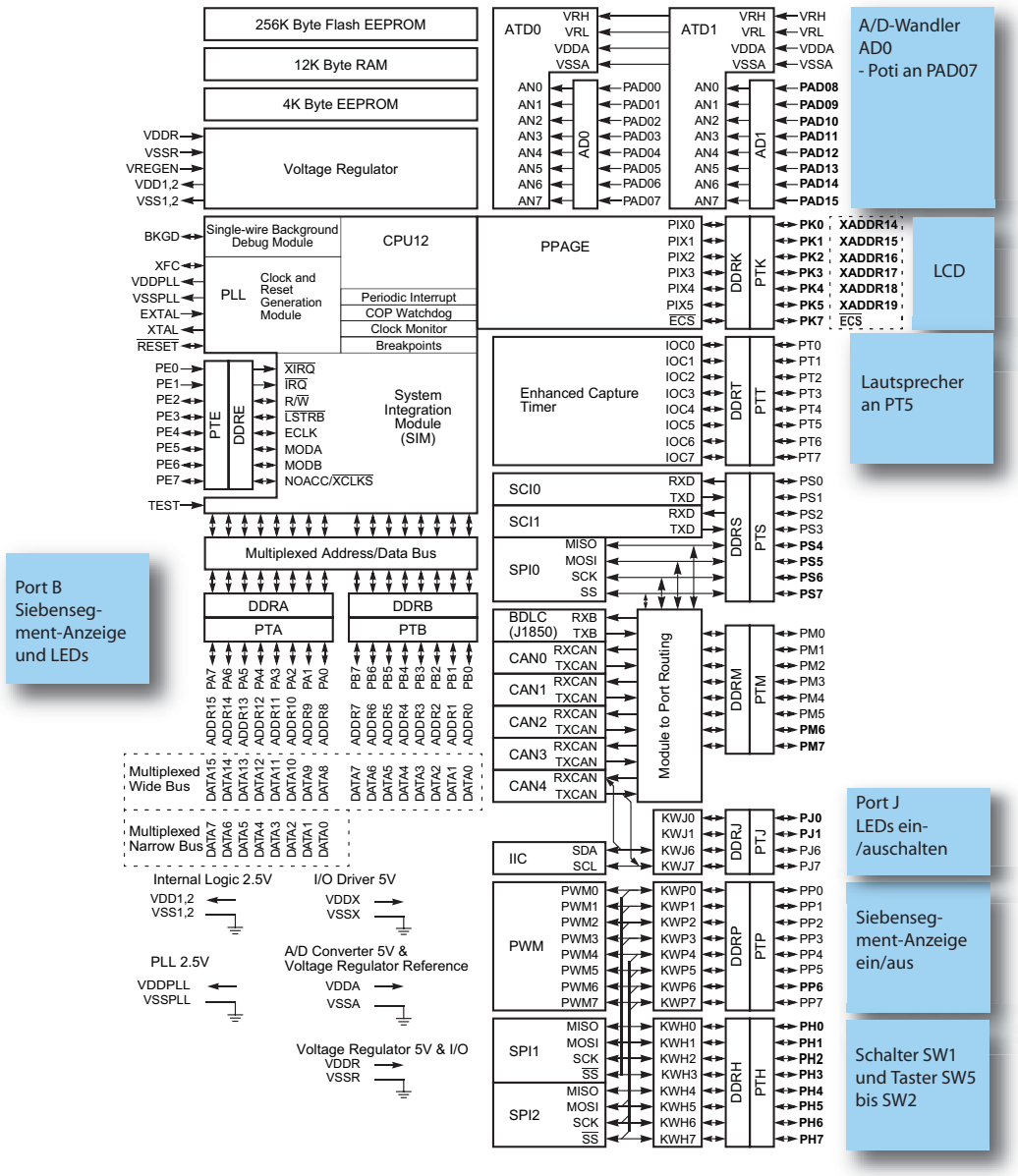


Abbildung B-3: Device-Übersicht

Index