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*.

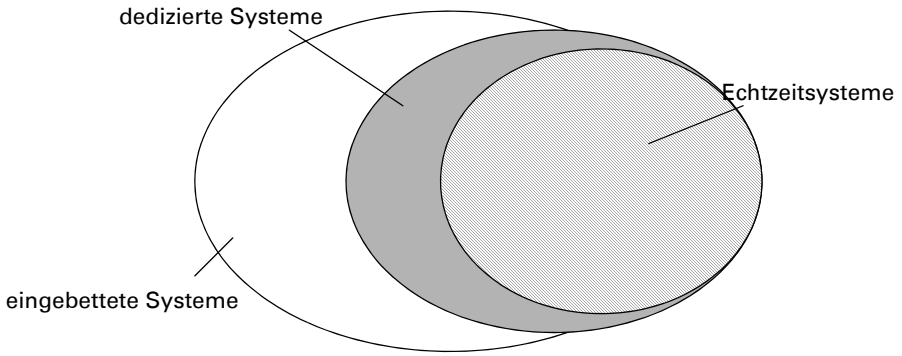


Bild 1.1: Abgrenzung eingebettete, dedizierte und Echtzeitsysteme

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

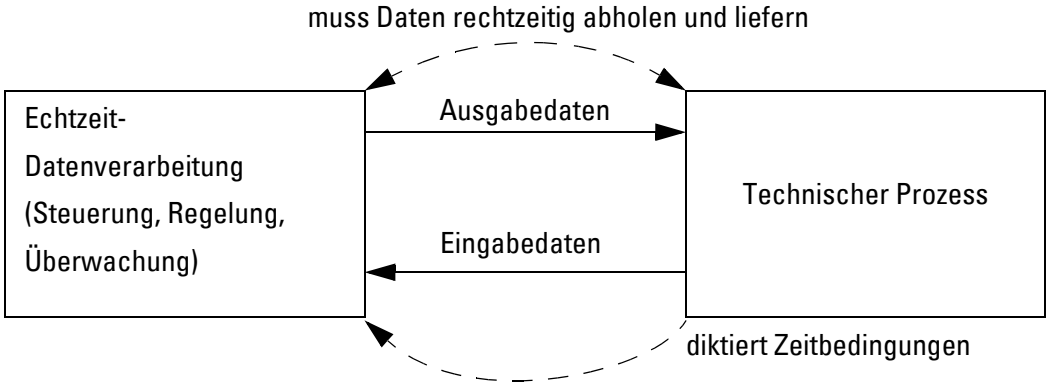
- Nicht-Echtzeitsysteme: *logische Korrektheit* bedeutet Korrektheit
- Echtzeitsysteme: *logische Korrektheit plus zeitliche Korrektheit* bedeutet Korrektheit

1.7.1 Rechtzeitigkeit

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

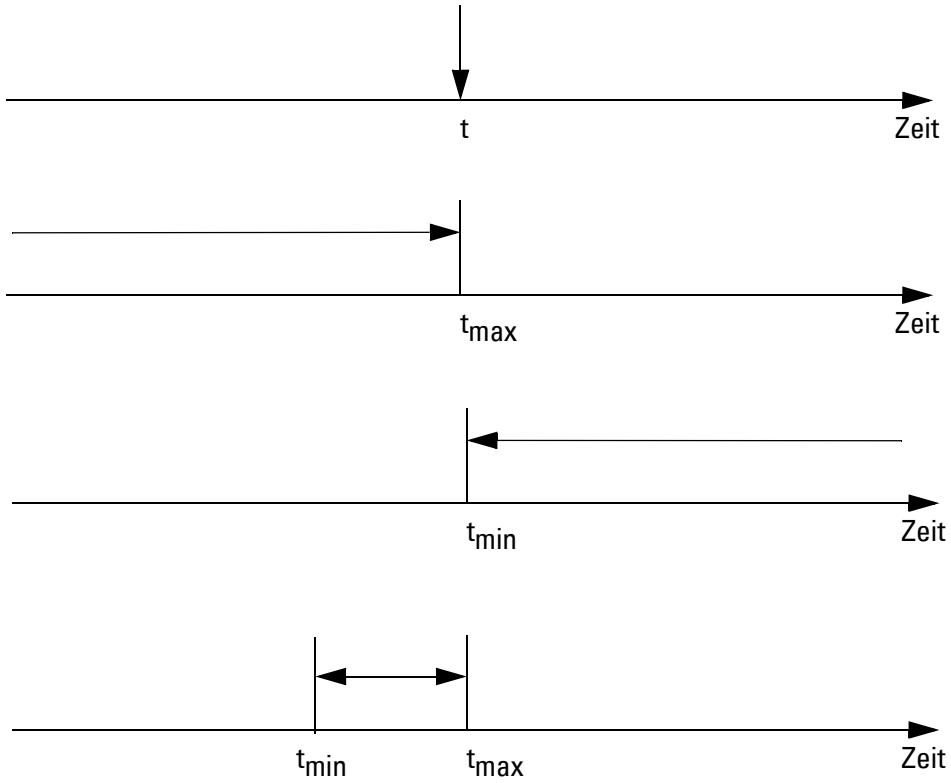
Erfordert indirekt auch die rechtzeitige Abholung von Eingangsdaten.

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)



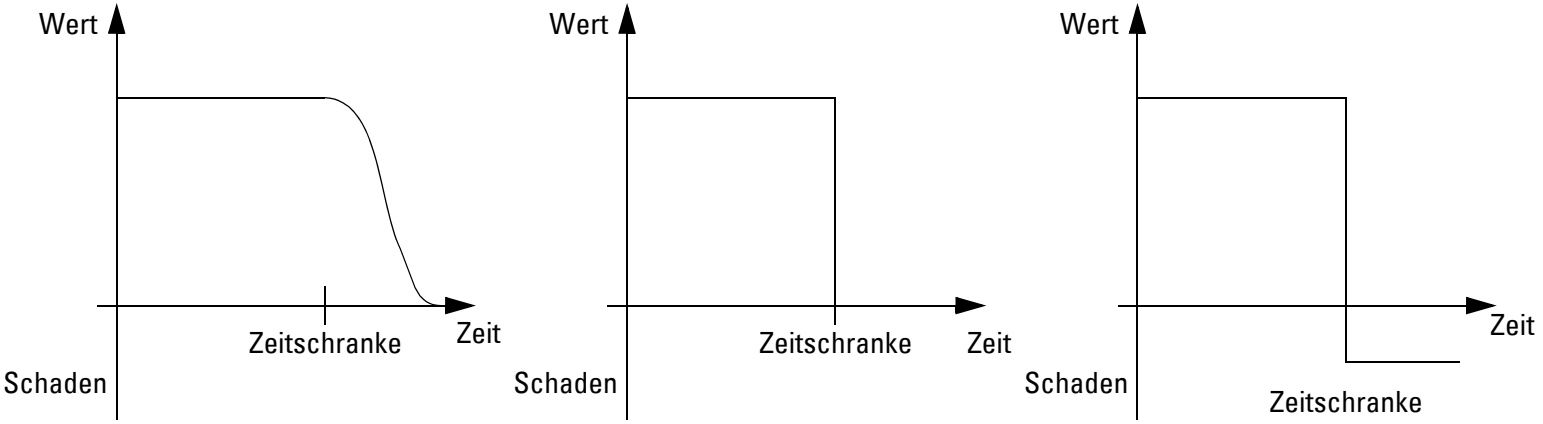
Unterteilung der *Zeitbedingungen* in

- *periodische* Zeitbedingungen
- *aperiodische* Zeitbedingungen

Weitere Unterteilung der *Zeitbedingungen* in

- *absolute* Zeitbedingungen
- *relative* Zeitbedingungen

Definitionen von harten, festen und weichen Echtzeitbedingungen:



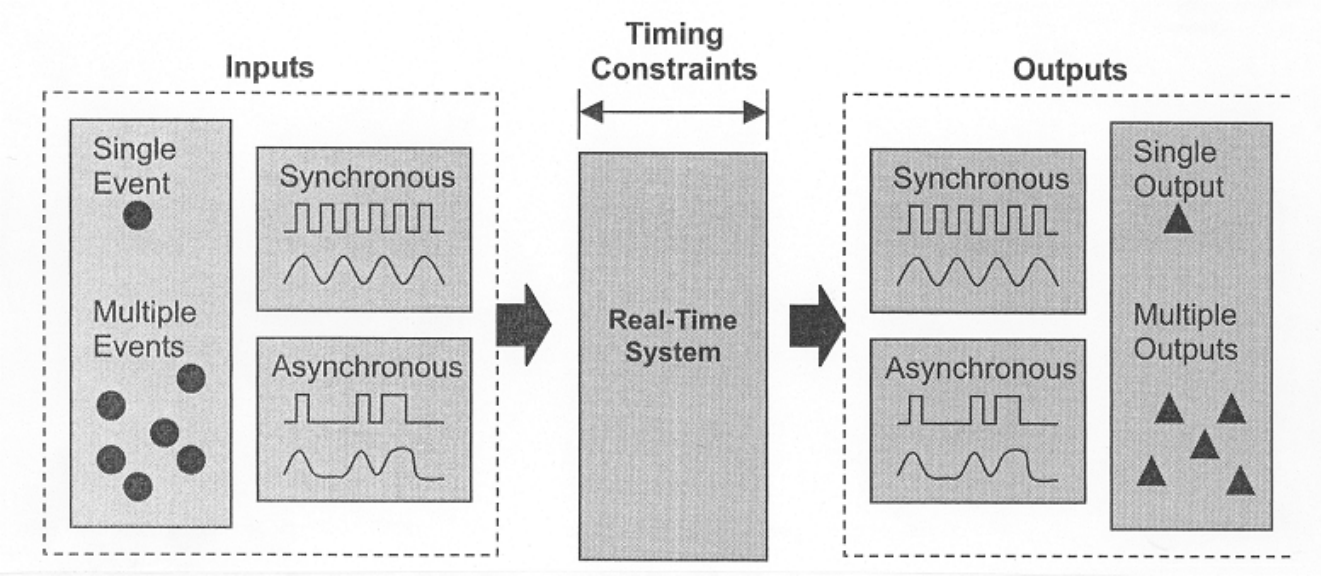
weich *fest* *hart*

Drei verschiedene Ansätze für Kriterien:

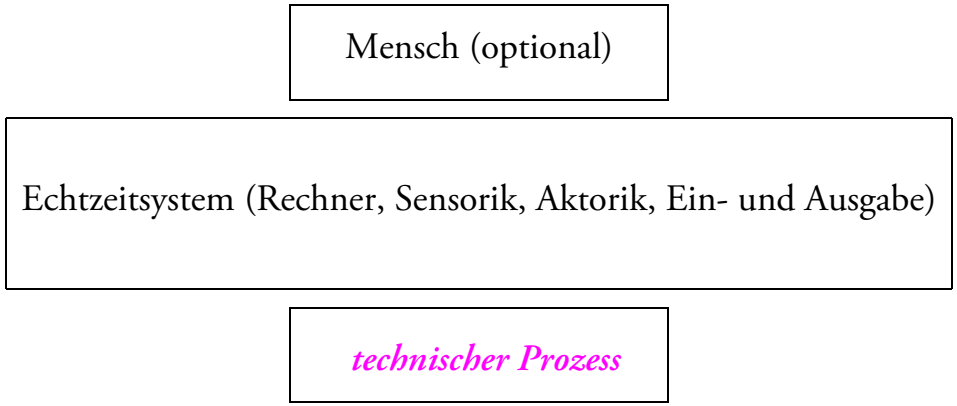
- nach Kritikalität (entsteht Schaden oder nicht)*; Beispiel: Airbag, Videoprozessor*
- nach Nützlichkeit (wie nützlich sind zu späte Ergebnisse)*; Problem hier: die Nützlichkeitsfunktion*
- nach zulässiger Häufigkeit von Zuspätkommen*; z.B. muss in 99,999% aller Fälle Zeitschranke einhalten*

1.8 Modell eines Echtzeitsystems

Ein einfaches Modell:



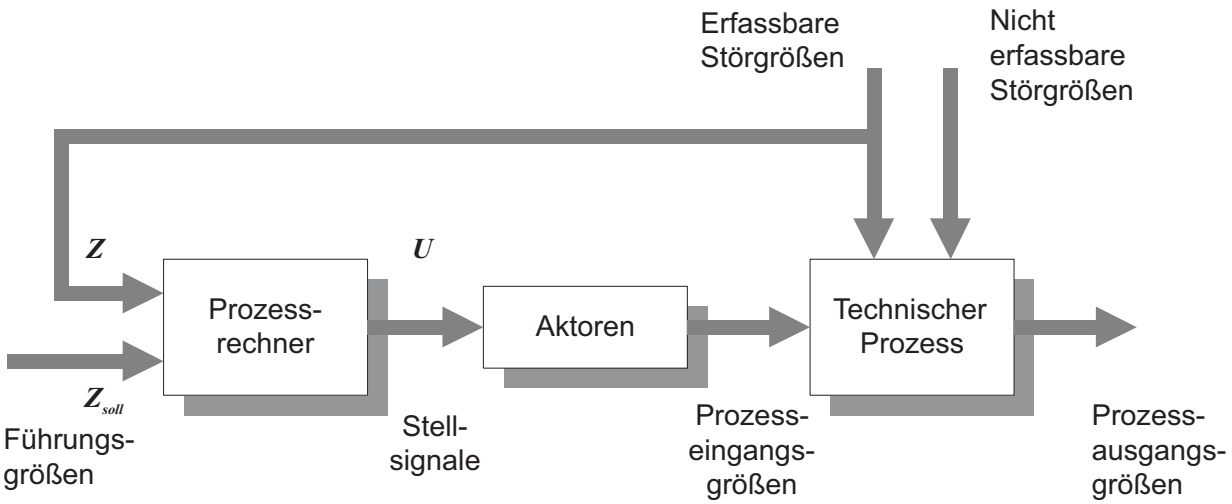
Andere Sicht:



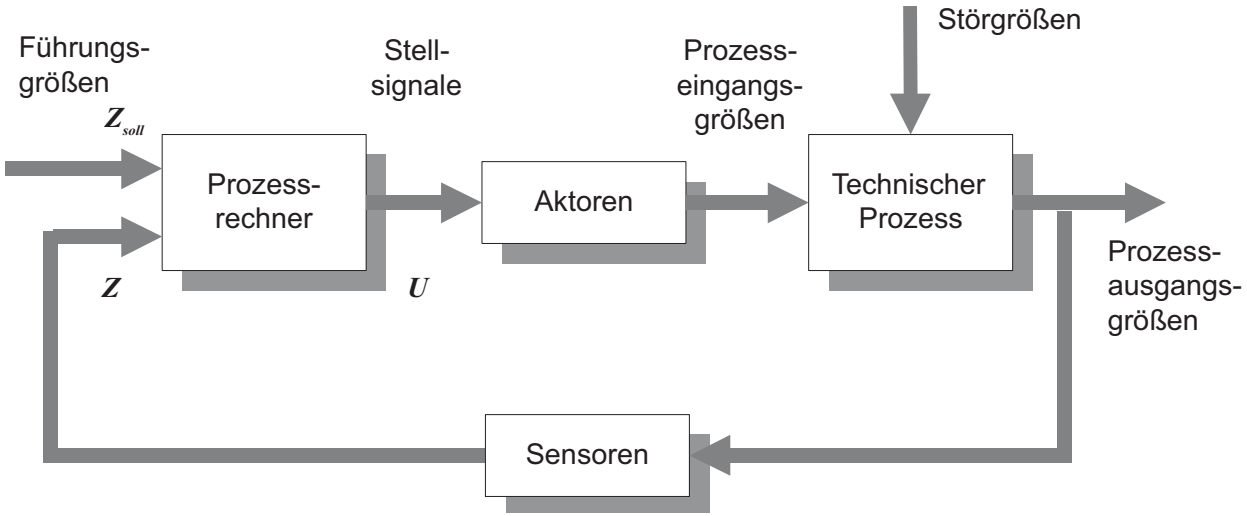
Echtzeitsysteme werden benutzt, um technische Prozesse zu

- *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.

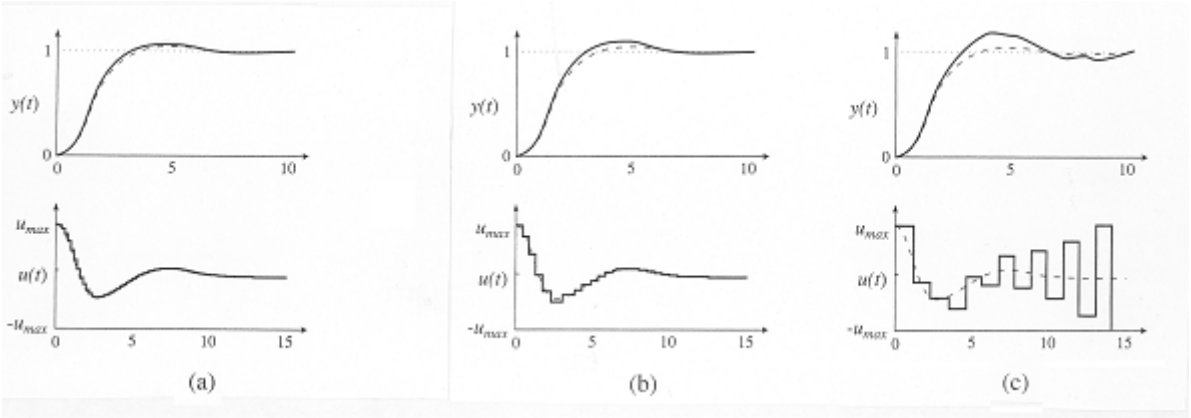
Prinzip der *Steuerung*:



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

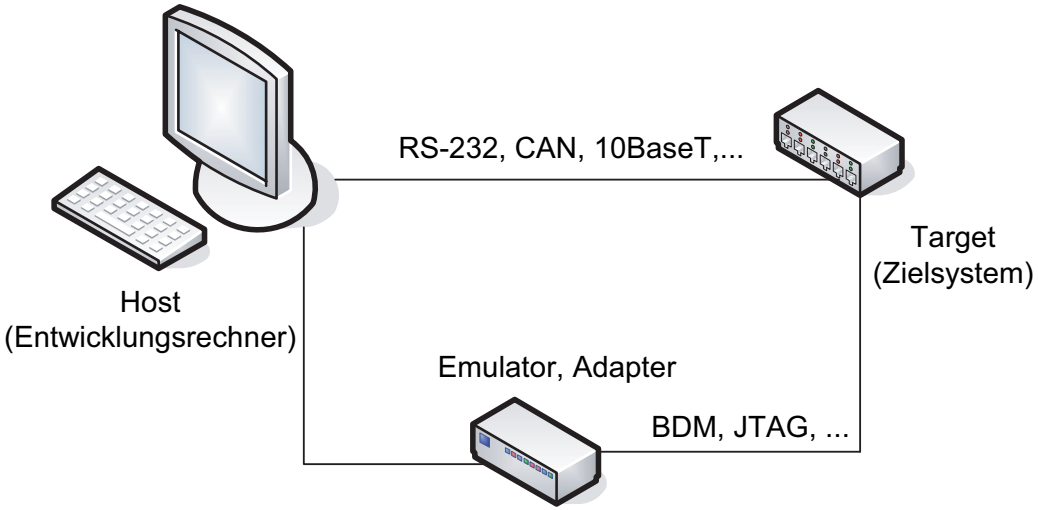


Abbildung 2-1: Entwicklungs- und Zielsystem

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

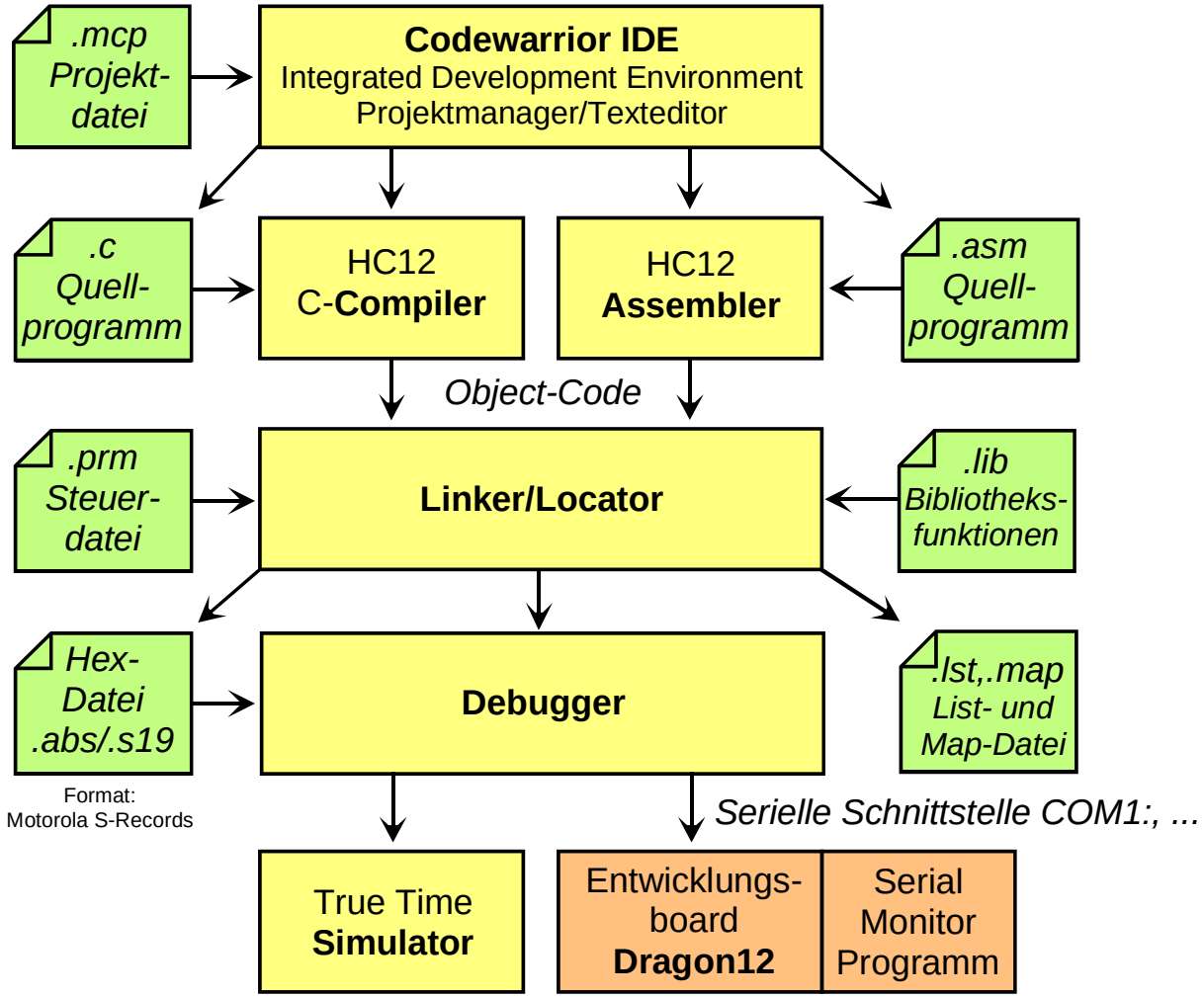
Standard-Entwicklungsrechner stellen heute USB-Anschlüsse und Ethernet-Anschlüsse sowie mit Einschränkungen RS-232-Schnittstellen zur Verfügung. Um JTAG-, BDM-, CAN- oder Emulator-Schnittstellen zu benutzen, ist ***zusätzliche Adapter-Hardware*** erforderlich.

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-

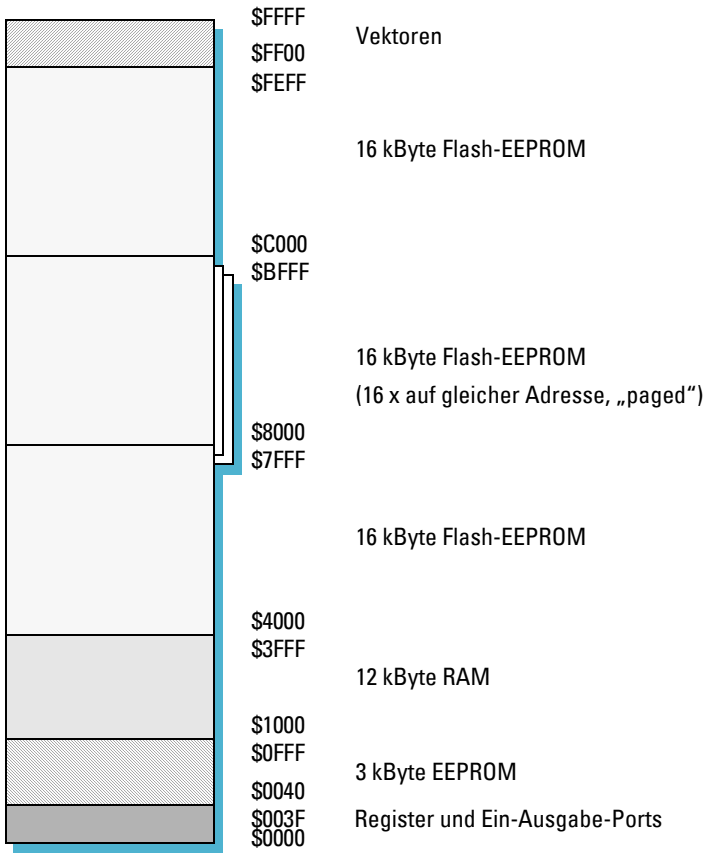
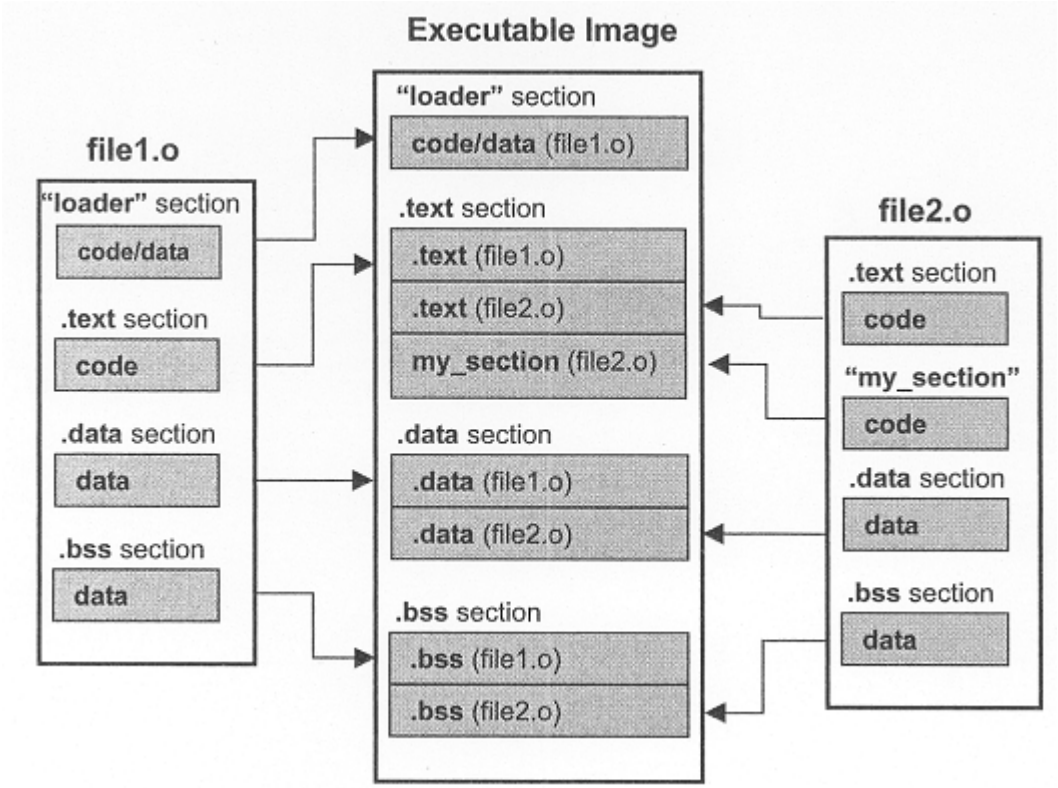


Abbildung 2-2: Memory-Map für den Freescale-Microcontroller 68HCS12

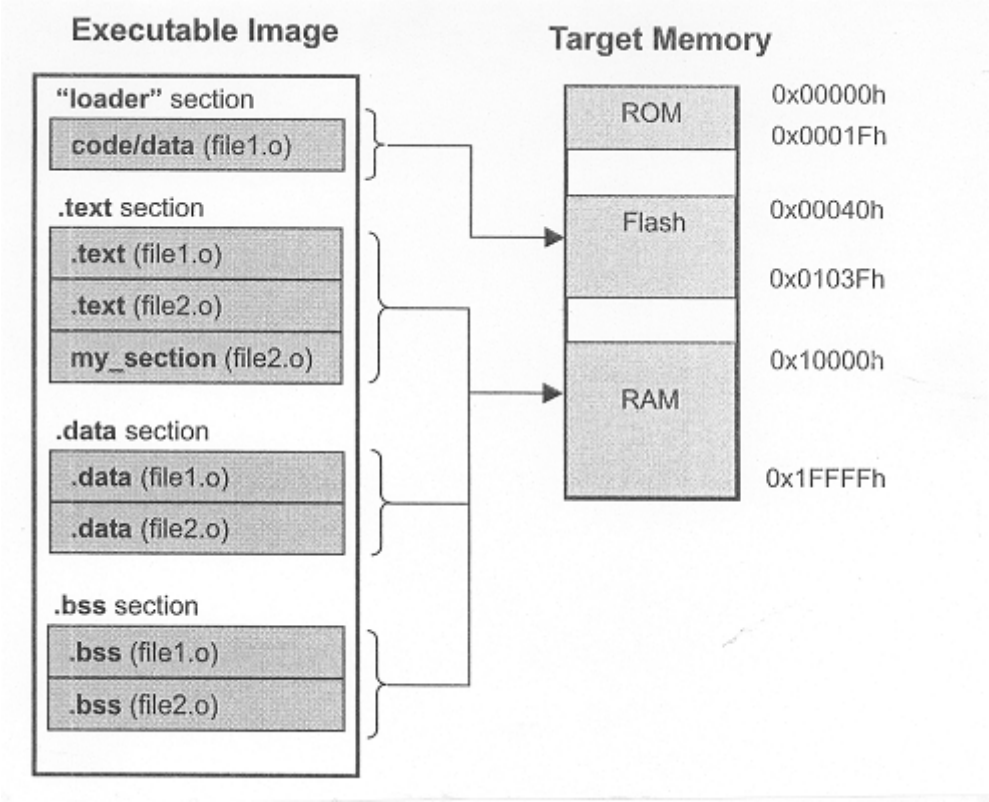
sung „Computerarchitektur 3“.

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.

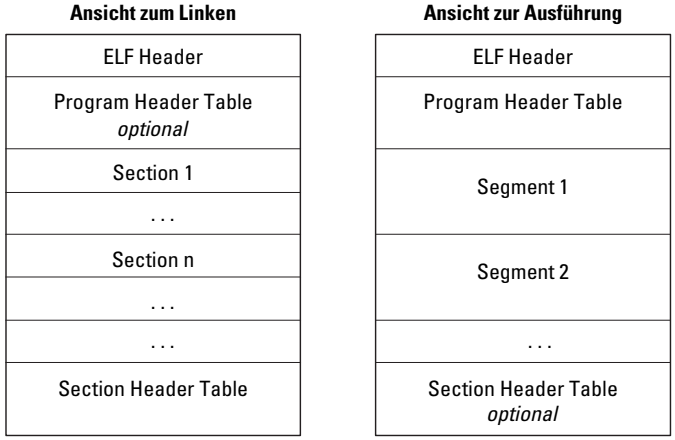


Bild 2.4: ELF-Dateistruktur

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:

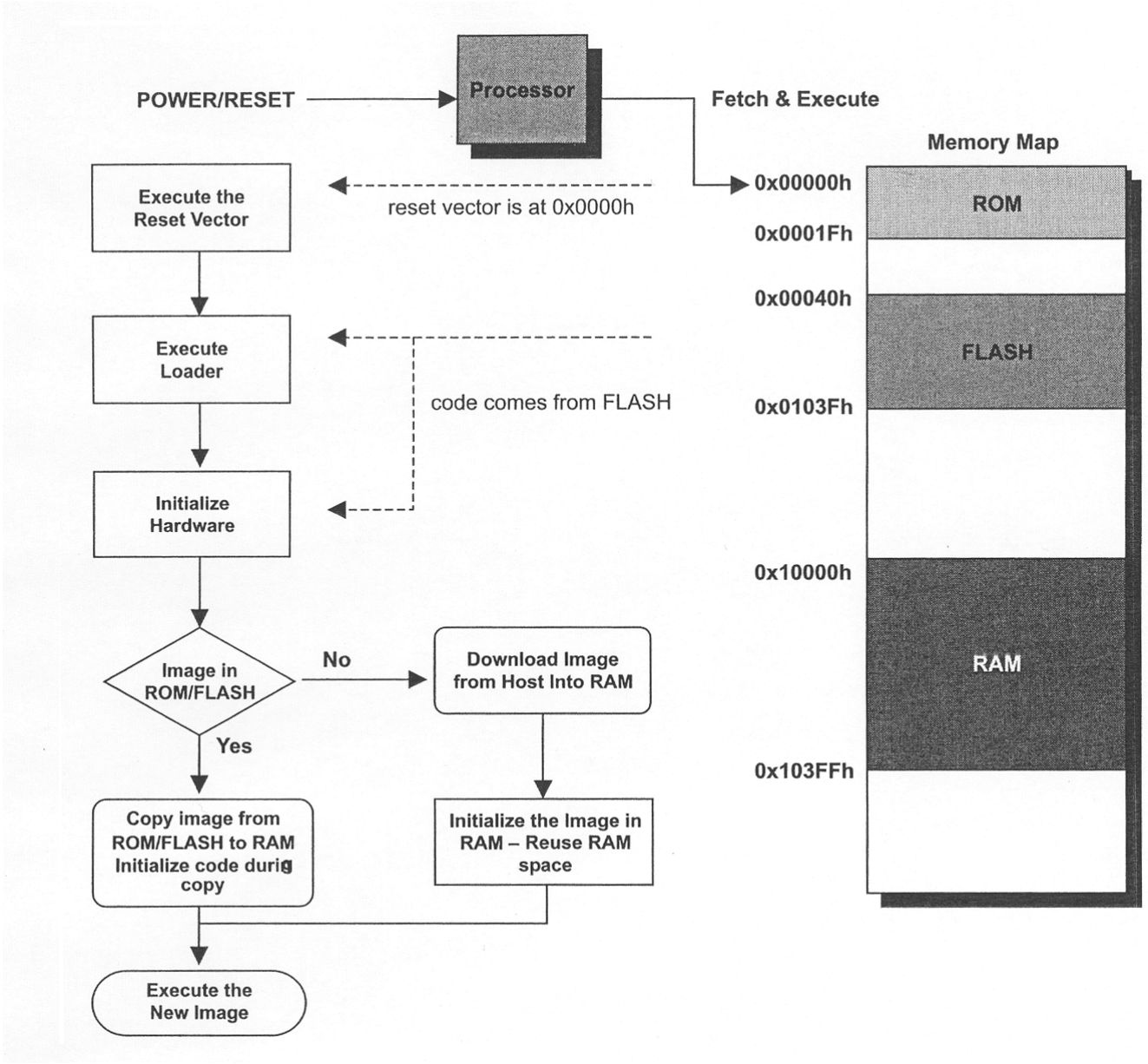
- Wenn Betriebssystemsystem vorhanden: Betriebssystem laden.

Schritt 3:

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

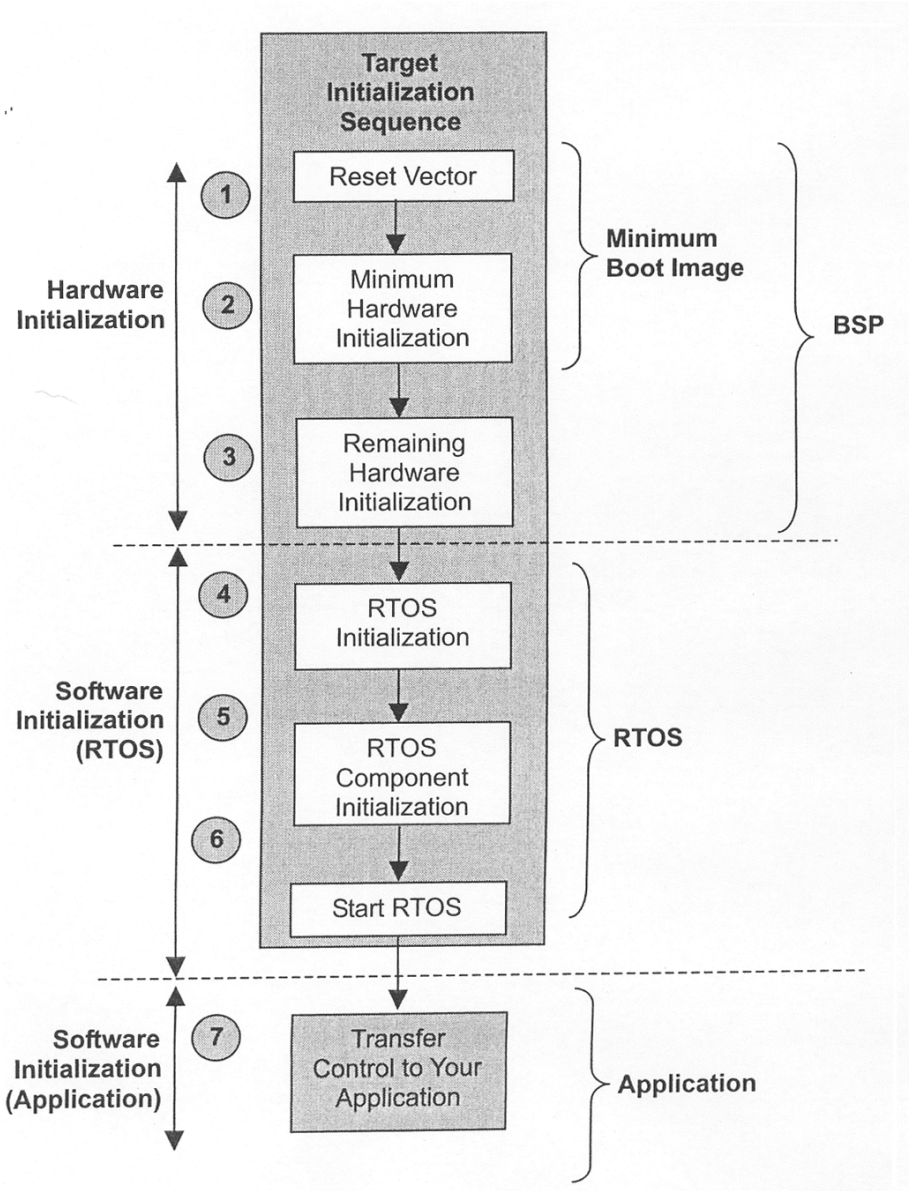
Schritt 4:

- 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

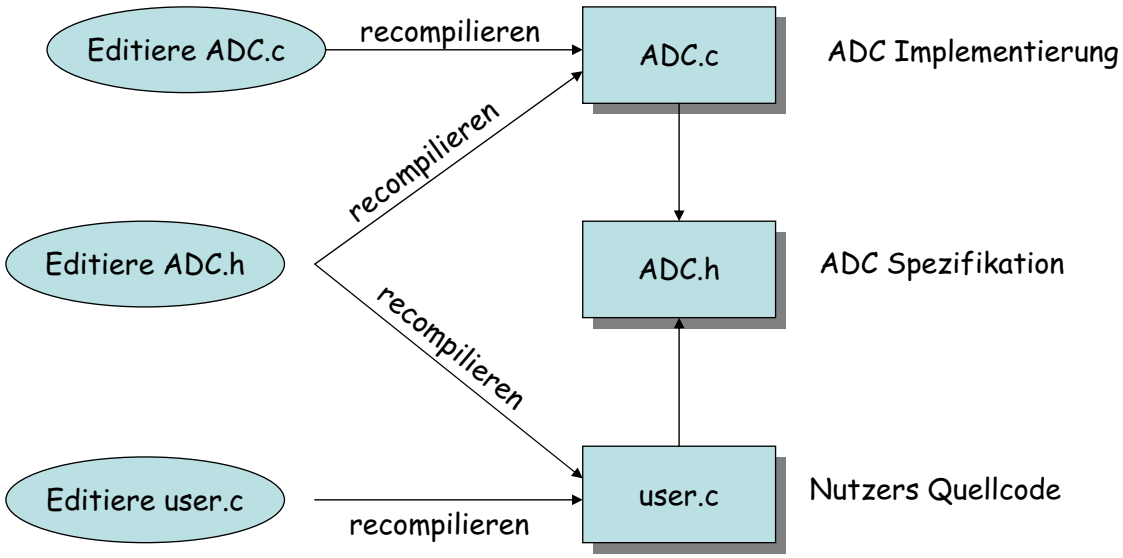
Weitere Regeln der Automotive-Industrie: MISRA (http://www.misra.org.uk). Dokument ist leider nicht frei verfügbar, muss man kaufen. Hier folgt meine persönliche Bestenliste.

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

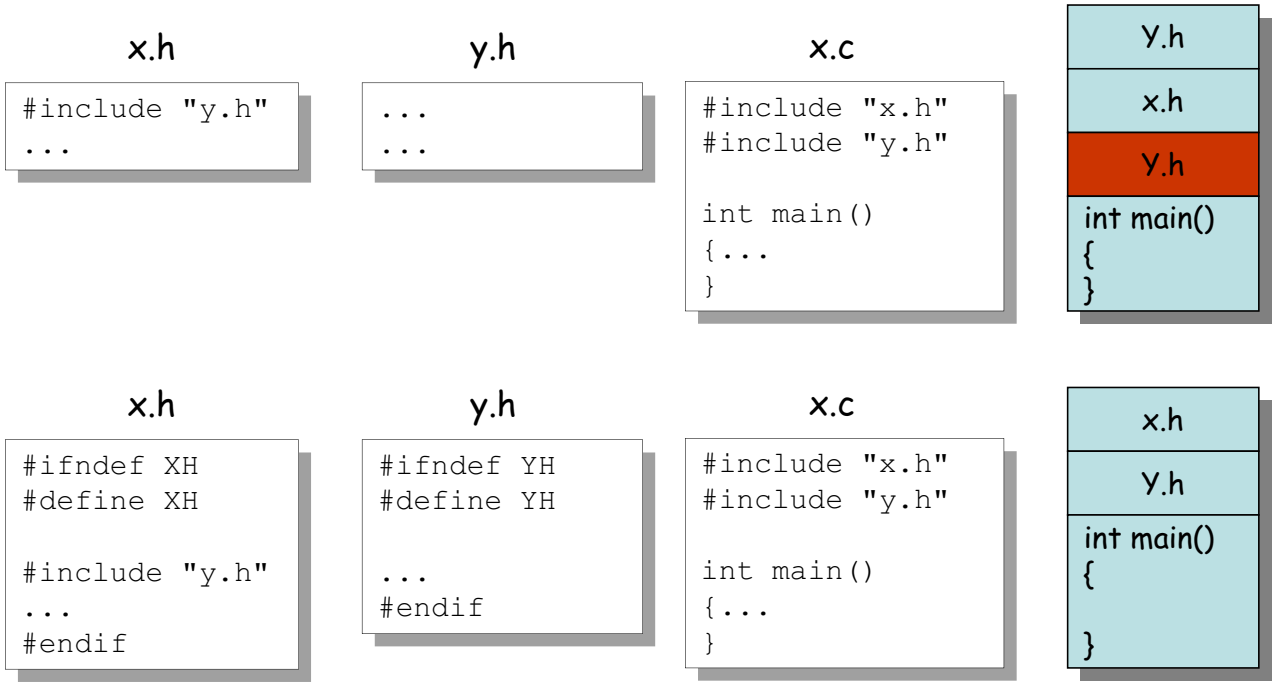
Siehe Darstellung. Wird bei einer IDE meist automatisch gehandhabt.



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

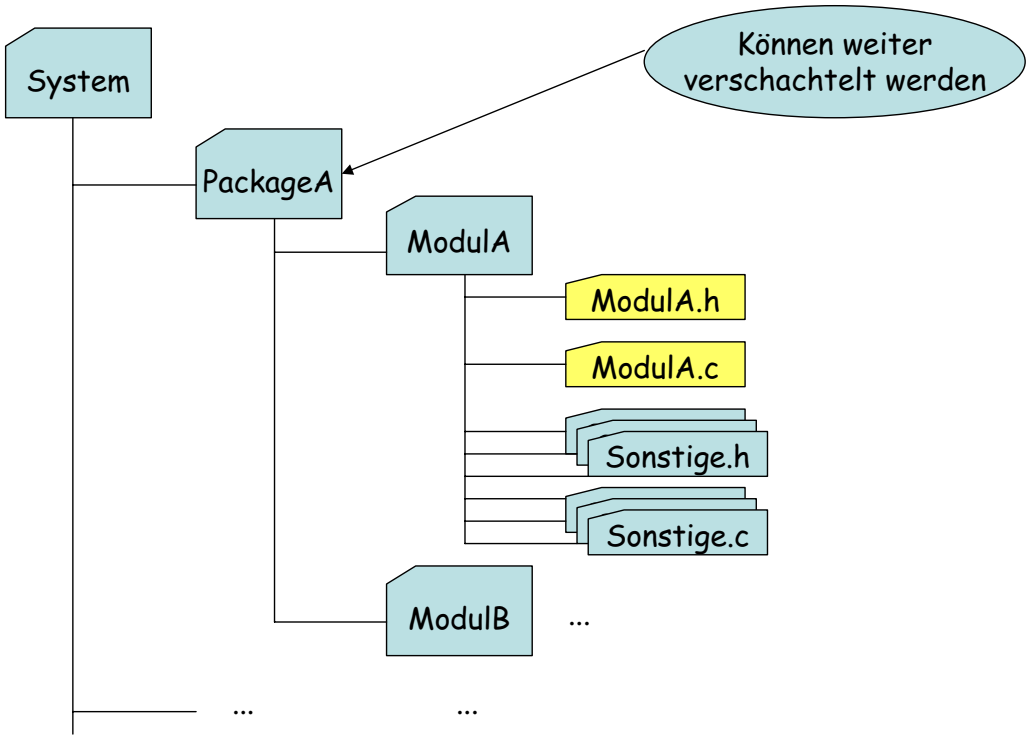
```
next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

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.