

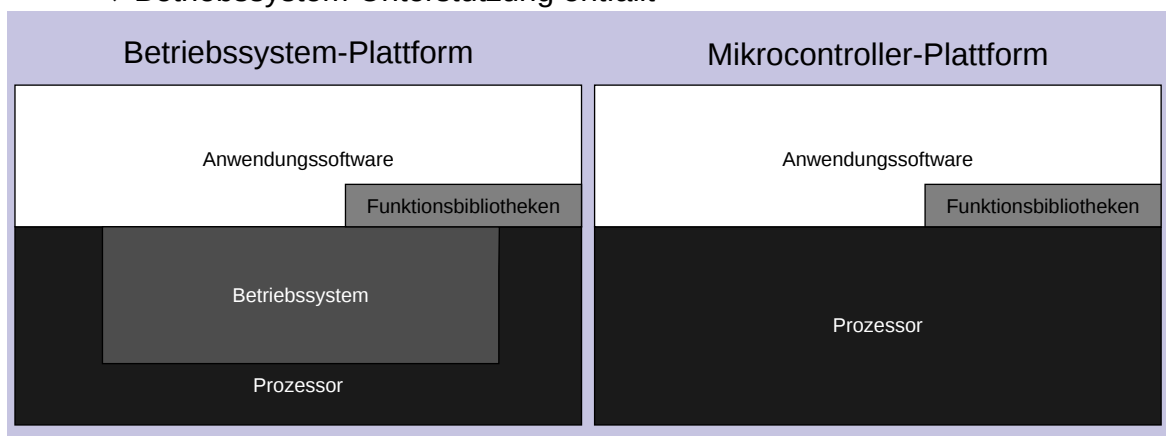
# J Mikrocontroller-Programmierung

## J.1 Überblick

- Mikrocontroller im Gegensatz zu Betriebssystem-Plattform
  - Prozessor am Beispiel AVR-Mikrocontroller
  - Speicher
  - Peripherie
- Programmausführung
  - Programm laden
  - starten
  - Fehler zur Laufzeit
- Interrupts
  - Grundkonzepte
  - Nebenläufigkeit
  - Umgang mit Nebenläufigkeit

## J.2 Mikrocontroller vs. Betriebssystem-Plattform

- Entscheidende Unterschiede:
  - ◆ Betriebssystem-Unterstützung entfällt



- ◆ Prozessor bietet in der Regel weniger / andere Funktionalität
  - kein virtueller Speicher
  - kein Speicherschutz
  - einfachere Peripherie-Ansteuerung

# 1 Betriebssystemunterstützung

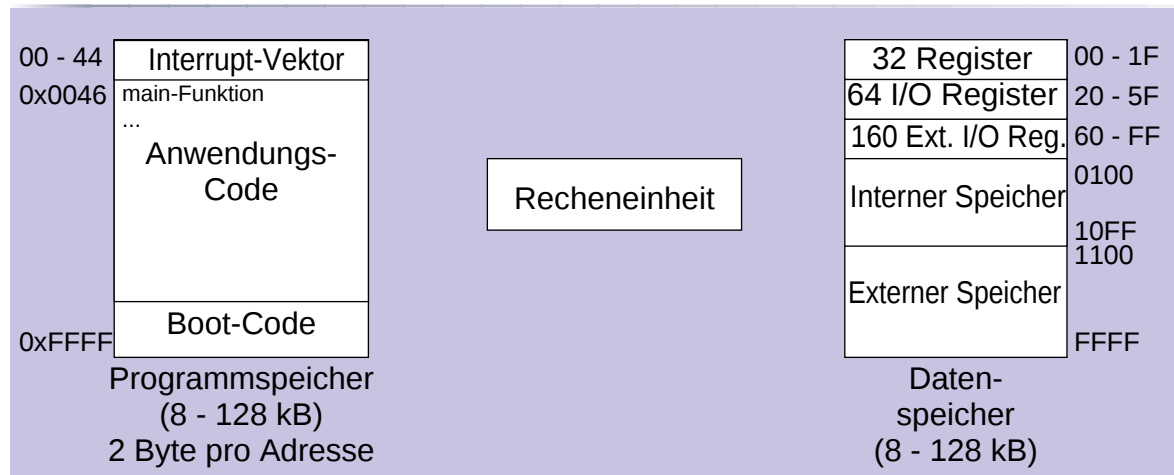
- Prozesse als Ausführungsumgebung für Programme
  - ◆ Erzeugen von Prozessen, Starten von Programmen
  - ◆ Ausführung mehrerer Prozesse quasi-gleichzeitig
  - ◆ Überwachung der Prozesse (z. B. beim Speicherzugriff), notfalls Abbrechen
- Dateisystem zur Abstraktion von Speichermedien und Peripheriegeräten
  - ◆ Abwicklung der Interaktion mit der Peripherie
    - Ansteuerung der Controller
    - Bearbeitung von Interrupts
  - ◆ Bereitstellen einer einheitlichen Schnittstelle (read/write) für Anwendungsprogramme
- Mechanismen zur Kommunikation zwischen Prozessen
  - ◆ gemeinsame Speicherbereiche
  - ◆ Nachrichtenaustausch
  - ◆ Signale

# 2 Mikrocontroller-Umgebung

- Programm läuft "nackt" auf der Hardware
  - ➔ Compiler und Binder müssen bereits ein vollständiges Programm erzeugen
    - kein dynamisches Binden zur Ladezeit
    - keine Betriebssystemunterstützung zur Laufzeit
  - ◆ Funktionalität muss entweder vom Anwender programmiert werden oder in Form von Funktionsbibliotheken zum Programm dazugebunden werden
  - ◆ Umgang mit "lästigen Programmierdetails" (z. B. bestimmte Bits setzen) wird durch Makros erleichtert
- Es wird genau ein Programm ausgeführt
  - Programm kann zur Laufzeit "niemanden stören"
  - Fehler betreffen nur das Programm selbst
  - keine Schutzmechanismen notwendig
    - ➔ ABER: Fehler ohne direkte Auswirkung werden leichter übersehen

## J.3 Beispiel: AVR-Mikrocontroller (ATmega-Serie)

### 1 Architektur



- Getrennter Speicher für Programm (Flash-Speicher) und Daten (SRAM)
- Register des Prozessors und Register für Ein-/Ausgabe-Schnittstellen sind in Adressbereich des Datenspeichers eingebettet

### 1 Architektur(2)

- Peripherie-Bausteine werden über I/O-Register angesprochen bzw. gesteuert (Befehle werden in den Bits eines I/O-Registers kodiert)
  - mehrere Timer (Zähler, deren Geschwindigkeit einstellbar ist und die bei einem bestimmten Wert einen Interrupt auslösen)
  - Ports (Gruppen von jeweils 8 Anschlüssen, die auf 0V oder  $V_{CC}$  gesetzt, bzw. deren Zustand abgefragt werden kann)
  - Output Compare Modulator (OCM) (zur Pulsweitenmodulation)
  - Serial Peripheral Interface (SPI)
  - Synchrone/Asynchrone serielle Schnittstelle (USART)
  - Analog-Comparator
  - A/D-Wandler
  - EEPROM (zur Speicherung von Konfigurationsdaten)

## 2 In Speicher eingebettete Register (memory mapped registers)

- Je nach Prozessor sind Zugriffe auf I/O-Register auf zwei Arten realisiert:
  - ◆ spezielle Befehle (z. B. in, out bei x86)
  - ◆ in den Adressraum eingebettet, Zugriff mittels Speicheroperationen
- Beim den meisten Mikrocontrollern sind die Register in den Speicher eingebettet
- Zugriffe auf die entsprechende Speicherstelle werden auf das entsprechende Register umgeleitet
- ➔ sehr einfacher und komfortabler Zugriff

## 3 I/O-Ports

- Ein I/O-Port ist eine Gruppe von meist 8 Anschluss-Pins und dient zum Anschluss von digitalen Peripheriegeräten
- Die Pins können entweder als Eingang oder als Ausgang dienen
  - ◆ als Ausgang konfiguriert, kann man festlegen, ob sie eine logische "1" oder eine logische "0" darstellen sollen
    - Ausgang wird dann entsprechend auf  $V_{CC}$  oder GND gesetzt
  - ◆ als Eingang konfiguriert, kann man den Zustand abfragen
- Manche I/O Pins können dazu genutzt werden einen Interrupt (IRQ = Interrupt Request) auszulösen (externe Interrupt-Quelle)
- Die meisten Pins können alternativ eine Spezialfunktion übernehmen, da sie einem integrierten Gerät als Ein- oder Ausgabe dienen können
  - ◆ z. B. dienen die Pins 0 und 1 von Port E (ATmega128) entweder als allgemeine I/O-Ports oder als RXD und TXD der seriellen Schnittstelle

## 4 Programme laden

- generell bei Mikrocontrollern mehrere Möglichkeiten
- ▲ Programm ist schon da (ROM)
- ▲ Bootloader-Programm ist da und liest Anwendung über serielle Schnittstelle ein und speichert sie im Programmspeicher ab
- ▲ spezielle Hardware-Schnittstelle
  - "jemand anderes" kann auf Speicher zugreifen
  - Beispiel: JTAG
    - spezielle Hardware-Komponente im AVR-Chip, die Zugriff auf die Speicher hat und mit der man über spezielle PINs kommunizieren kann

## 5 Programm starten

- Reset bewirkt Ausführung des Befehls an Adresse 0x0000
  - dort steht ein Sprungbefehl auf die Speicheradresse der main-Funktion
  - alternativ: Sprungbefehl auf Adresse des Bootloader-Programms, Bootloader lädt Anwendung und springt dann auf main-Adresse

## 6 Fehler zur Laufzeit

- Zugriff auf ungültige Adresse
  - ◆ es passiert nichts:
    - Schreiben geht in's Leere
    - Lesen ergibt zufälliges Ergebnis
- ungültige Operation auf nur-lesbare / nur-schreibbare Register/Speicher
  - hat keine Auswirkung

## J.4 Interrupts

### 1 Motivation

- An einer Peripherie-Schnittstelle tritt ein Ereignis auf
  - Spannung wird angelegt
  - Zähler ist abgelaufen
  - Gerät hat Aufgabe erledigt (z. B. serielle Schnittstelle hat Byte übertragen, A/D-Wandler hat neuen Wert vorliegen)
  - Gerät hat Daten für die Anwendung bereit stehen (z. B. serielle Schnittstelle hat Byte empfangen)
- ? wie bekommt das Programm das mit?
  - Zustand der Schnittstelle regelmäßig überprüfen (= **Polling**)
  - Schnittstelle meldet sich von sich aus beim Prozessor und unterbricht den Programmablauf (= **Interrupt**)

### 1 Motivation (2)

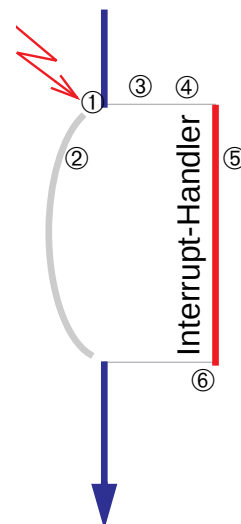
- Polling vs. Interrupts: Vor und Nachteile
  - + Pollen erfolgt **synchron** zum Programmablauf, Programm ist in dem Moment auf das Ereignis vorbereitet
  - Interrupts unterbrechen den Programmablauf irgendwo (**asynchron**), sie könnten in dem Augenblick stören
    - ➔ durch die Interrupt-Bearbeitungsteht **Nebenläufigkeit**
  - Pollen erfolgt explizit im Programm und meistens umsonst — Rechenzeit wird verschwendet
  - + Interrupts melden sich nur, wenn tatsächlich etwas zu erledigen ist
  - + Interrupt-Bearbeitung ist in einer Funktion kompakt zusammengefasst
  - Polling-Funktionalität ist in den normalen Programmablauf eingestreut — und hat mit der "eigentlichen" Funktionalität dort meist nichts zu tun

## 2 Implementierung

- typischerweise mehrere Interrupt-Quellen
- Interrupt-Vektor
  - ◆ Speicherbereich (Tabelle), der für jeden Interrupt Informationen zur Bearbeitung enthält
    - Maschinenbefehl  
(typischerweise ein Sprungbefehl auf eine Adresse, an der eine Bearbeitungsfunktion steht)  
oder
    - Adresse einer Funktion (**Interrupt-Handler**)
  - ◆ feste Position im Speicher — ist im Prozessorhandbuch nachzulesen
- Maskieren von Interrupts
  - Bit im Prozessor-Statusregister schaltet den Empfang aller Interrupts ab
  - zwischenzeitlich eintreffende Interrupts werden gepuffert (nur einer!)
  - die Erzeugung einzelner Interrupts kann am jeweiligen Gerät unterbunden werden

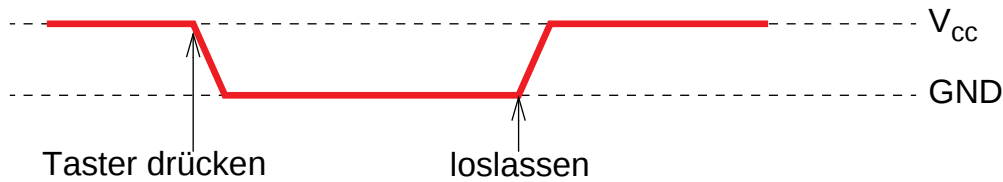
## 3 Ablauf

- ① Gerät löst Interrupt aus, Ablauf des Anwendungsprogramms wird unmittelbar unterbrochen
- ② weitere Interrupts werden deaktiviert
- ③ aktuelle Position im Programm und Registerinhalte werden im Datenspeicher gesichert
- ④ Eintrag im Interrupt-Vektor ermitteln
- ⑤ Befehl wird ausgeführt bzw. Funktion aufrufen
- ⑥ am Ende der Bearbeitungsfunktion bewirkt ein Befehl "Return from Interrupt" die Fortsetzung des Anwendungsprogramms und die Reaktivierung der Interrupts



## 4 Pegel- und Flanken-gesteuerte Interrupts

### ■ Beispiel: Signal eines Tasters



### ■ Flanken-gesteuert

- Interrupt wird durch die Flanke (= Wechsel des Pegels) ausgelöst
- welche Flanke einen Interrupt auslöst kann bei manchen Prozessoren konfiguriert werden

### ■ Pegel-gesteuert

- solange ein bestimmter Pegel anliegt (hier Pegel = GND) wird immer wieder ein Interrupt ausgelöst

## J.5 Nebenläufigkeit

### 1 Überblick

#### ■ Definition von Nebenläufigkeit:

*zwei Programmausführungen sind nebenläufig, wenn für zwei einzelne Befehle a und b aus beiden Ausführungen nicht feststeht, ob a oder b tatsächlich zuerst ausgeführt wird*

#### ■ Nebenläufigkeit tritt auf

- bei Interrupts
- bei parallelen Abläufen (gleichzeitige Ausführung von Code in einem Mehrprozessorsystem mit Zugriff auf den gleichen Speicher)
- bei quasi-parallelen Abläufen (wenn ein Betriebssystem verschiedenen Prozesse den Prozessor jeweils für einen Zeitraum zuteilt und ihn nach Ablauf der Zeit wieder entzieht)

#### ■ Problem:

- was passiert, wenn die nebenläufigen Ausführungen auf die gleichen Daten im Speicher zugreifen?

## 2 Nebenläufigkeit durch Interrupts

- Interrupts unterbrechen Anwendungsprogramme "irgendwo"
- Interrupts haben Zugriff auf den gleichen Speicher
- Szenario:
  - eine Lichtschranke soll Fahrzeuge zählen und alle 10 Sekunden soll der Wert ausgegeben werden

```
int a;

int main() {
    long i;
    while(1) {
        for (i=0; i<2000000; i++)
            /* Zählen dauert 10 Sek. */;
        print(a);
        a=0;
    }
}
```

```
/* Lichtschranken-
   Interrupt */
int count() {
    a++;
}
```

## 2 Nebenläufigkeit durch Interrupts (2)

- Auf C-Ebene führt die Interrupt-Behandlung nur einen Befehl aus: a++
  - nur scheinbar ein Befehl
  - auf Maschinencode-Ebene (Bsp. AVR) sieht die Sache anders aus

```
...
    print(a);
    a=0;
...
```

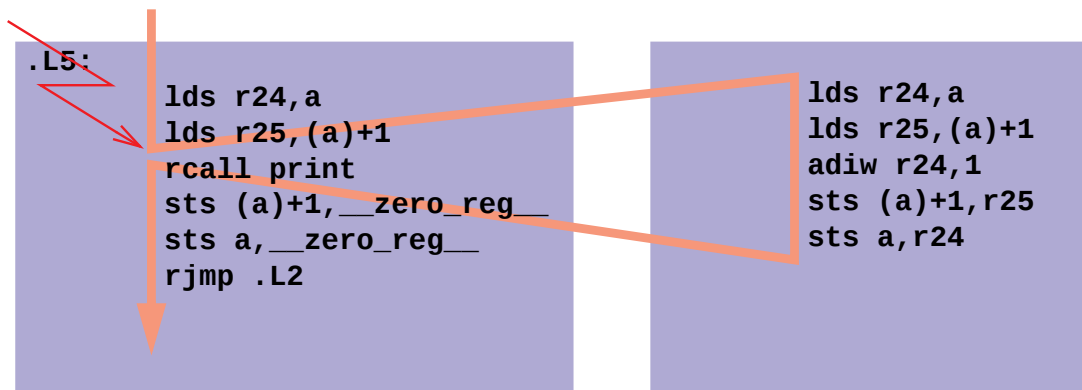
```
int count() {
    a++;
}
```

```
...
.L5:
    lds r24,a
    lds r25,(a)+1
    rcall print
    sts (a)+1,__zero_reg__
    sts a,__zero_reg__
    rjmp .L2
...
```

```
...
    lds r24,a
    lds r25,(a)+1
    adiw r24,1
    sts (a)+1,r25
    sts a,r24
...
```

## 2 Nebenläufigkeit durch Interrupts (3)

- Annahme: Interrupt trifft folgendermaßen ein:



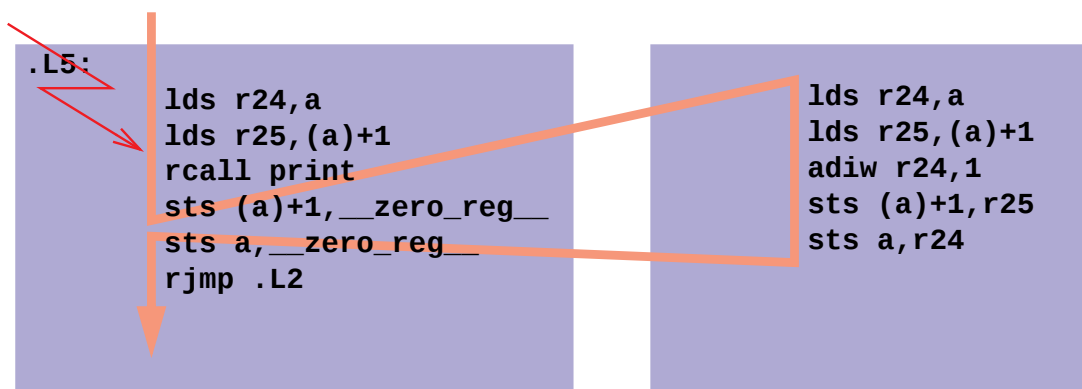
- Folge: ein Fahrzeug wird nicht gezählt

- Details des Szenarios zeigen mehrere Problemstellen:

- int-Wert wird in zwei Schritten in zwei Register geladen (long: 4 Register)
- Operationen erfolgen in Registern, danach wird in Speicher zurückgeschrieben

## 2 Nebenläufigkeit durch Interrupts (4)

- Annahme: Interrupt trifft folgendermaßen ein:



- Folge: möglicherweise werden 255 Fahrzeuge zuviel gezählt
- Variable a ist auf 2 Register verteilt → a = 0 nicht atomar  
zuerst wird obere Hälfte auf 0 gesetzt
  - falls a++ im Interrupt-Handler a zufällig von 255 auf 256 zählt  
→ Bitüberlauf vom "unteren" in's "obere" Register
  - nach Interrupt wird nur noch untere Hälfte auf 0 gesetzt → a = 256

## 2 Nebenläufigkeit durch Interrupts (5)

- weiteres Problem bei Zugriff auf globale Variablen:
  - ◆ AVR stellt 32 Register zur Verfügung
  - ◆ Compiler optimiert Code und vermeidet Speicherzugriffe wenn möglich
    - Variablen werden möglichst in Registern gehalten
  - ◆ Registerinhalte werden bei Interrupt gesichert und am Ende restauriert
    - Änderungen der Interrupt-Funktion an einer Variablen gehen beim Restaurieren der Register wieder verloren

- Lösung für dieses Problem:

- ◆ Compiler muss Variablen vor jedem Zugriff aus dem Speicher laden und anschließend zurückschreiben
  - Attribut `volatile`

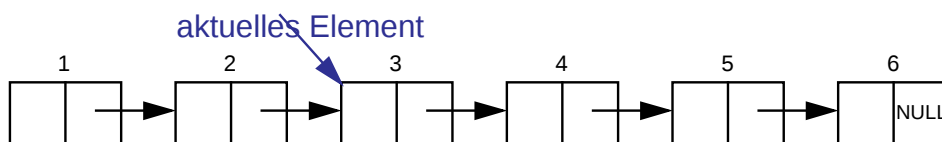
```
volatile int a;
```

- ◆ Nachteil: Code wird umfangreicher und langsamer
  - nur einsetzen wo unbedingt notwendig!

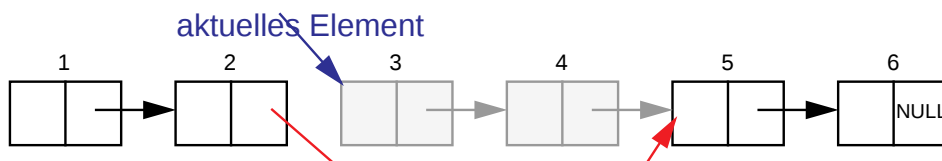
## 3 Nebenläufigkeitsprobleme allgemein

- Zugriff auf gemeinsame Daten ist bei nebenläufigen Ausführungen generell kritisch
  - selbst bei einfachen Variablen (siehe vorheriges Beispiel)
  - Problem bei komplexeren Datenstrukturen (z. B. Einketten einer Struktur in verkettete Liste)
    - noch gravierender: Datenstruktur kann völlig zerstört werden

- Beispiel: Programm läuft durch eine verkettete Liste



- ◆ Interrupt-Handler oder parallel laufendes Programm entfernt Elemente 3 und 4 und gibt den Speicher dieser Elemente frei



## 4 Umgang mit Nebenläufigkeitsproblemen

- Gemeinsame Daten möglichst vermeiden
  - Interrupt-Funktionen sollten weitgehend auf eigenen Daten arbeiten
  - Parallele Abläufe sollten ebenfalls möglichst eigene Datenbereiche haben
- Kommunikation zwischen Anwendungsabläufen erfordert aber oft gemeinsame Daten
  - solche Daten sollten deutlich hervorgehoben werden  
z. B. durch entsprechenden Namen
 

**`volatile int INT_zaeher;`**

    - betrifft nur globale Variablen
    - lokale Variablen sind unkritisch  
(nur in der jeweiligen Funktion sichtbar)
  - Zugriff auf solche Daten sollte in der Anwendung möglichst begrenzt sein  
(z. B. nur in bestimmten Funktionen, gemeinsames Modul mit Interrupt-Handlern, vgl. Kap. D.9-3)

## 4 Umgang mit Nebenläufigkeitsproblemen (2)

- Zugriffskonflikte mit Interrupt-Handlern verhindern
  - ◆ das Programm muss vor kritischen Zugriffen auf gemeinsame Daten Interrupts sperren
    - Beispiel AVR:  
Funktionen `cli()` (blockiert alle Interrupts)  
und `sei()` (erlaubt Interrupts)
  - ◆ in dieser Zeit können Interrupts verloren gehen
  - ➡ Zeitraum von Interruptsperrern muss möglichst kurz bleiben!
  - ◆ es kann sinnvoll sein, nur Interrupts des Geräts zu sperren, dessen Handler auch auf die kritischen Daten zugreift  
(hängt von Details der Hardware ab!)
- Zugriffskonflikte bei parallelen Abläufen
  - ◆ spezielle atomare Maschinenbefehle  
(z. B. test-and-set oder compare-and-swap bei Intel-Architekturen)
  - ◆ Software-Synchronisation (lock-Variablen, Semaphore, etc.)
  - ◆ Kommunikation mittels Nachrichten statt gemeinsamer Daten

## J.6 Programmierung eingebetteter Systeme

- Vorteile von C in eingebetteten Systemen:
  - leichter Zugriff auf die Hardware
  - gute Kontrolle über die verwendeten Ressourcen (Speicher, CPU)
- Probleme mit C:
  - stark eingeschränkte Laufzeitüberprüfungen
  - ISO C-Standard oft nicht eindeutig  
Verhalten ist implementationsabhängig oder undefiniert
  - ➔ in C kann man viele Fehler machen (v. a. Anfänger)

### 1 Misra C

- 1998 von der Misra entwickelt  
(Misra = Motor Industry Software Reliability Association)
- Richtlinien für den Gebrauch von C in sicherheitskritischen Systemen
- ursprünglich Automobilbereich, seit 2004 neue verallgemeinerte Version
- Regeln, die bestimmte Konstrukte von C verbieten  
z. B. Zeigeroperationen

## 2 Java für eingebettete Systeme?

- mögliche Vorteile:
  - objektorientiert
  - typsicher

## 3 Objektorientierte Programmierung

- Zusammenfassung von Funktionen und ihren Daten
  - ➔ Strukturierung
- Vorteile / Nachteile
  - + Zusammengehörendes kommt zusammen
  - + kaum noch globale Variablen
  - + höhere Abstraktion möglich
  - weniger Kontrolle über Ressourcenverbrauch
  - ➔ Werkzeuge müssen für eingebettete Systeme optimiert sein

## 4 Typsichere Sprachen

- Beispiele: Java, .NET-Sprachen (z. B. C#)
- Grundprinzip:
  - allen Werten ist immer ein Typ zugeordnet
  - Umwandlung in einen anderen Typ ist nicht ohne weiteres möglich
- Umwandlung muss möglicherweise zur Laufzeit geprüft werden
  - ➔ keine Zeigeroperationen
- Automatische Speicherbereinigung möglich
- Typsystem zur Laufzeit nötig
  - ➔ meist interpretiert

## 4 Typsichere Sprachen (2)

- Vorteile / Nachteile:
  - mehr Ressourcenbedarf durch Laufzeitüberprüfungen
  - schlechtes Laufzeitverhalten bei Interpretation
  - + Interpretation ermöglicht leichte Erweiterbarkeit zur Laufzeit (dynamisches Nachladen von Klassen)
  - + Typsicherheit ermöglicht die Kapselung der Anwendung ohne Hardwareunterstützung (vertrauenswürdiger Compiler!)
  
- Für tief eingebettete Systeme ist Ressourcenbedarf wichtig
  - starke Einschränkung der dynamischen Erweiterbarkeit
  - compilieren zu statischem Programm

## 5 Java für eingebettete Systeme

- Java Micro-Edition (J2ME):
  - schlanke JVM und angepasste Klassenbibliothek
  - als J2ME/CLDC-MIDP auf den meisten Handys
  - als J2ME/CDC auf einigen PDAs
  
- JavaCard: stark abgespeckte JVM
  - für Chipkarten

## 6 Modellbasiertes Design

- MDA: Model Driven Architecture
- Beschreibung der Anwendung in Form eines Modells auf einer abstrakten Ebene
- Konkretes Programm wird von einem Werkzeug erstellt
- Vorteile/Nachteile
  - + architekturunabhängig
  - + Konzentration auf die Funktionalität der Anwendung
  - +/- Detailwissen zur Implementierung in den Werkzeugen
  - Werkzeug für die Zielplattform wird benötigt
- Beispiel: Labview, Simulink

## 7 Familienbasierter Ansatz

- Programme haben meist eine Vielzahl von Funktionen
- Nicht immer wird alles benötigt
  - ➔ nur das auswählen was für die aktuelle Aufgabe benötigt wird
- Vorteile / Nachteile
  - + Ressourceneinsparung, da nur benötigte Funktionalität vorhanden
  - + durch Werkzeugunterstützung ist trotzdem eine komfortable Codeverwaltung möglich
  - Abhängigkeiten zwischen den Funktionen müssen erfasst und berücksichtigt werden

# K letzte Infos / Organisatorisches

---

- Musterklausur ab Ende Juli verfügbar (über die Web-Seite)
- 2 Fragestunden jeweils in der Woche vor der Klausur
  - ◆ 1. Termin: voraussichtlich am 2.8.
  - ◆ 2. Termin steht noch nicht fest, vermutlich in KW40
- Fragen im Forum diskutieren!
- Details und Termine auf den Webseiten zur Vorlesung