Systemnahe Programmierung in C (SPiC)

Teil C Systemnahe Softwareentwicklung

Daniel Lohmann, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2011

http://www4.informatik.uni-erlangen.de/Lehre/SS11/V_SPIC

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 μC-Systemarchitektur

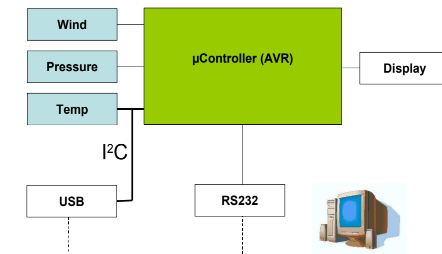
Softwareentwurf

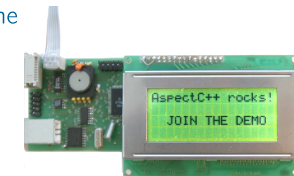
- Softwareentwurf: Grundsätzliche Überlegungen über die Struktur eines Programms **vor** Beginn der Programmierung
  - Ziel: Zerlegung des Problems in beherrschbare Einheiten
- Es gibt eine Vielzahl von Softwareentwurfs-Methoden
  - Objektorientierter Entwurf [↪ GDI, IV]
    - Stand der Kunst
    - Dekomposition in Klassen und Objekte
    - An Programmiersprachen wie C++ oder Java ausgelegt
  - Top-Down-Entwurf / Funktionale Dekomposition
    - Bis Mitte der 80er Jahre fast ausschließlich verwendet
    - Dekomposition in Funktionen und Funktionsaufrufe
    - An Programmiersprachen wie Fortran, Cobol, Pascal oder C orientiert

Systemnahe Software wird oft (noch) mit **Funktionaler Dekomposition** entworfen und entwickelt.

Beispiel-Projekt: Eine Wetterstation

- Typisches eingebettetes System
  - Mehrere Sensoren
    - Wind
    - Luftdruck
    - Temperatur
  - Mehrere Aktoren (hier: Ausgabegeräte)
    - LCD-Anzeige
    - PC über RS232
    - PC über USB
  - Sensoren und Aktoren an den μC angebunden über verschiedene Bussysteme
    - I^2C
    - RS232



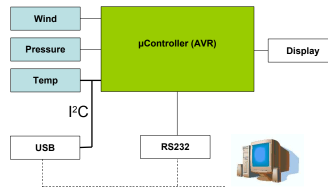


Wie sieht die **funktionale Dekomposition** der Software aus?

Funktionale Dekomposition: Beispiel

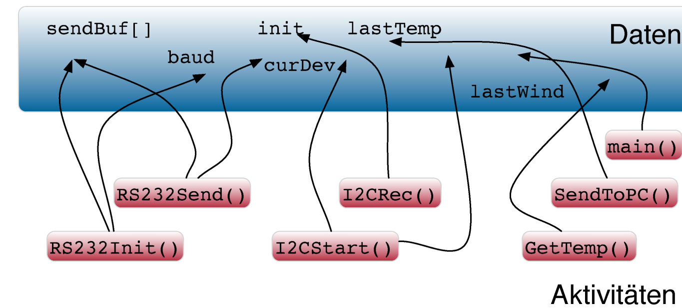
Funktionale Dekomposition der Wetterstation (Auszug):

1. Sensordaten lesen
   1.1 Temperatursensor lesen
      1.1.1 I^2C-Datenübertragung intiieren
      1.1.2 Daten vom I^2C-Bus lesen
   1.2 Drucksensor lesen
   1.3 Windsensor lesen
2. Daten aufbereiten (z. B. glätten)
3. Daten ausgeben
   3.1 Daten über RS232 versenden
      3.1.1 Baudrate und Parität festlegen (einmalig)
      3.1.2 Daten schreiben
   3.2 LCD-Display aktualisieren
4. Warten und ab Schritt 1 wiederholen



Funktionale Dekomposition: Probleme

- Erzielte Gliederung betrachtet nur die Struktur der Aktivitäten, nicht jedoch die die Struktur der Daten
- Gefahr: Funktionen arbeiten „wild“ auf einer Unmenge schlecht strukturierter Daten ⤳ mangelhafte Trennung der Belange



Funktionale Dekomposition: Probleme

- Erzielte Gliederung betrachtet nur die Struktur der Aktivitäten, nicht jedoch die die Struktur der Daten
- Gefahr: Funktionen arbeiten „wild“ auf einer Unmenge schlecht strukturierter Daten ⤳ mangelhafte Trennung der Belange

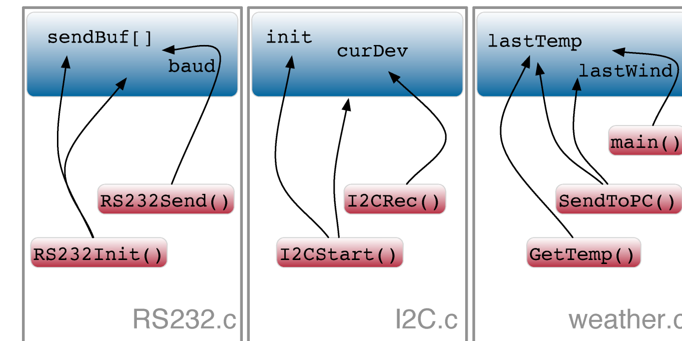
Prinzip der Trennung der Belange

Dinge, die **nichts miteinander** zu tun haben, sind auch **getrennt** unterzubringen!

Trennung der Belange (*Separation of Concerns*) ist ein **Fundamentalprinzip** der Informatik (wie auch jeder anderen Ingenieursdisziplin).

Lösung: Modularisierung

- Separation jeweils zusammengehöriger Daten und Funktionen in übergeordnete Einheiten ⤳ **Module**



Was ist ein Modul?

- **Modul** := (<*Menge von Funktionen*>, <*Menge von Daten*>, <*Schnittstelle*>) (↦ „`class`" in Java)
- Module sind größere Programmbausteine ↪ 9–1
  - Problemorientierte Zusammenfassung von Funktionen und Daten
    ⇝ Trennung der Belange
  - Ermöglichen die einfache Wiederverwendung von Komponenten
  - Ermöglichen den einfachen Austausch von Komponenten
  - Verbergen Implementierungsdetails (**Black-Box**-Prinzip)
    ⇝ Zugriff erfolgt ausschließlich über die Modulschnittstelle

Modul ↦ Abstraktion ↪ 4–1

- Die Schnittstelle eines Moduls abstrahiert
  - Von der tatsächlichen Implementierung der Funktionen
  - Von der internen Darstellung und Verwendung von Daten

Module in C [≠Java]

- In C ist das Modulkonzept nicht Bestandteil der Sprache, sondern rein idiomatisch (über Konventionen) realisiert ↪ 3–13
  - Modulschnittstelle ↦ `.h`-Datei (enthält Deklarationen ↪ 9–7)
  - Modulimplementierung ↦ `.c`-Datei (enthält Definitionen ↪ 9–3)
  - Modulverwendung ↦ `#include <Modul.h>`

```
void RS232Init( uint16_t br );
void RS232Send( char ch );
...
```

RS232.h: **Schnittstelle / Vertrag (öffentl.)**
Deklaration der bereitgestellten Funktionen (und ggf. Daten)

```
#include <RS232.h>
static uint16_t   baud = 2400;
static char       sendBuf[16];
...
void RS232Init( uint16_t br) {
  ...
  baud = br;
}
void RS232Send( char ch ) {
  sendBuf[...] = ch;
  ...
}
```

RS232.c: **Implementierung (nicht öffentl.)**
Definition der bereitgestellten Funktionen (und ggf. Daten)

Ggf. modulinterne Hilfsfunktionen und Daten (`static`)

Inklusion der eigenen Schnittstelle stellt sicher, dass der Vertrag eingehalten wird

Module in C – Export [≠Java]

- Ein C-Modul exportiert eine Menge von definierten Symbolen
  - Alle Funktionen und globalen Variablen (↦ „`public`" in Java)
  - Export kann mit `static` unterbunden werden (↦ „`private`" in Java)
    (↦ Einschränkung der Sichtbarkeit ↪ 10–3)
- Export erfolgt beim Übersetzungsvorgang (`.c`-Datei ⟶ `.o`-Datei)

Quelldatei (`foo.c`)

```
int a;                 // public
static int b;          // private

void f(void)           // public
{ ... }
static void g(int)     // private
{ ... }
```

Objektdatei (`foo.o`)

Symbole **a** und **f** werden exportiert.

Symbole b und g sind `static` definiert und werden deshalb nicht exportiert.

Module in C – Import [≠Java]

- Ein C-Modul importiert eine Menge nicht-definierter Symbole
  - Funktionen und globale Variablen, die verwendet werden, im Modul selber jedoch nicht definiert sind
  - Werden beim Übersetzen als unaufgelöst markiert

Quelldatei (`bar.c`)

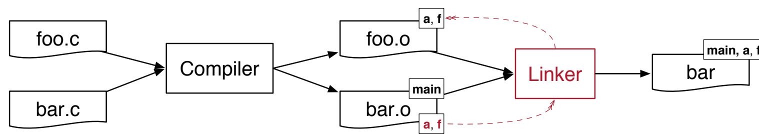
```
extern int a;       // declare
void f(void);       // declare

void main() {       // public
  a = 0x4711;       // use
  f();              // use
}
```

Objektdatei (`bar.o`)

Symbol **`main`** wird exportiert.
Symbole **a** und **f** sind unaufgelöst.

Module in C – Import (Forts.) [≠Java]

- Die eigentliche Auflösung erfolgt durch den Linker [↪ GDI, VI-158]



Linken ist nicht typsicher!

- Typinformationen sind in Objektdateien nicht mehr vorhanden
- Auflösung durch den Linker erfolgt **ausschließlich** über die **Symbolnamen** (Bezeichner)

⇝ Typsicherheit muss beim **Übersetzen** sichergestellt werden

⇝ Einheitliche Deklarationen durch gemeinsame Header-Datei

Module in C – Header [≠Java]

- Elemente aus fremden Modulen müssen deklariert werden
  - Funktionen durch normale Deklaration ↪ 9–7

```
void f(void);
```

  - Globale Variablen durch `extern`

```
extern int a;
```

> Das `extern` unterscheidet eine Variablendeklaration von einer Variablendefinition.

- Die Deklarationen erfolgen sinnvollerweise in einer Header-Datei, die von der Modulentwicklerin bereitgestellt wird
  - Schnittstelle des Moduls (↦ „`interface`" in Java)
    - Exportierte Funktionen des Moduls
    - Exportierte globale Variablen des Moduls
    - Modulspezifische Konstanten, Typen, Makros
    - Verwendung durch Inklusion (↦ „`import`" in Java)
  - Wird **auch vom Modul inkludiert**, um Übereinstimmung von Deklarationen und Definitionen sicher zu stellen (↦ „`implements`" in Java)

Module in C – Header (Forts.) [≠Java]

Modulschnittstelle: `foo.h`

```
// foo.h
#ifndef _FOO_H
#define _FOO_H

// declarations
extern int a;
void f(void);

#endif // _FOO_H
```

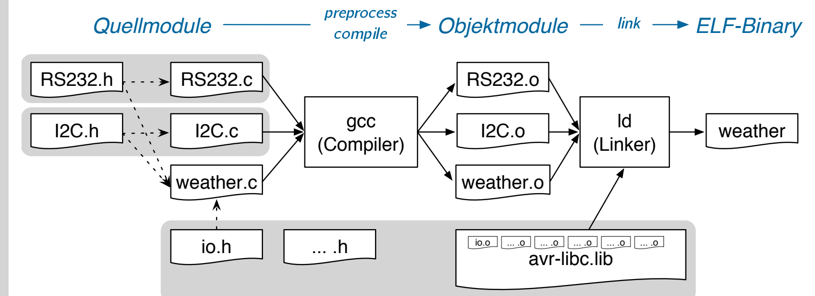
Modulimplementierung `foo.c`

```
// foo.c
#include <foo.h>


// definitions
int a;
void f(void){
  ...
}
```

Modulverwendung `bar.c` (vergleiche ↪ 12–9)

```
// bar.c
~~extern int a;~~
~~void f(void);~~
#include <foo.h>

void main() {
  a = 0x4711;
  f();
}
```

Zurück zum Beispiel: Wetterstation



- Jedes Modul besteht aus Header- und Implementierungsdatei(en)
  - `.h`-Datei definiert die Schnittstelle
  - `.c`-Datei implementiert die Schnittstelle, inkludiert `.h`-Datei, um sicherzustellen, dass Deklaration und Definition übereinstimmen
- Modulverwendung durch Inkludieren der modulspezifischen `.h`-Datei
- Das Ganze funktioniert entsprechend bei Bibliotheken