

Systemnahe Programmierung in C

12 Programmstruktur und Module

J. Kleinöder, D. Lohmann, V. Sieh

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2025

<http://sys.cs.fau.de/lehre/ss25>



- 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
 - 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



- 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
 - 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.



■ Typisches eingebettetes System

■ Mehrere Sensoren

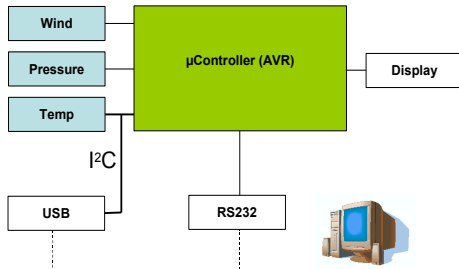
- Wind
- Luftdruck
- Temperatur

■ Mehrere Aktuatoren (hier: Ausgabegeräte)

- LCD-Anzeige
- PC über RS232
- PC über USB

■ Sensoren und Aktuatoren an den μC angebunden über verschiedene Bussysteme

- I²C
- RS232



Beispiel-Projekt: Eine Wetterstation

■ Typisches eingebettetes System

■ Mehrere Sensoren

- Wind
- Luftdruck
- Temperatur

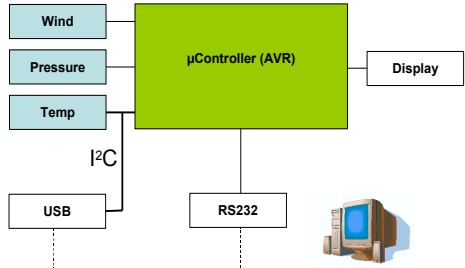
■ Mehrere Aktuatoren (hier: Ausgabegeräte)

- LCD-Anzeige
- PC über RS232
- PC über USB

■ Sensoren und Aktuatoren an den μC angebunden über verschiedene Bussysteme

- I²C
- RS232

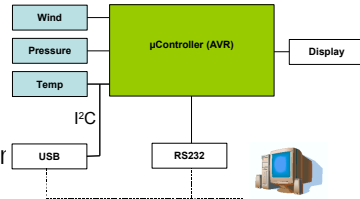
Wie sieht die **funktionale Dekomposition** der Software aus?



Funktionale Dekomposition: Beispiel

Funktionale Dekomposition der Wetterstation (Auszug):

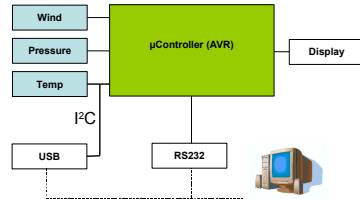
1. Sensordaten lesen
2. Daten aufbereiten (z. B. glätten)
3. Daten ausgeben
4. Warten und ab Schritt 1 wiederholen



Funktionale Dekomposition: Beispiel

Funktionale Dekomposition der Wetterstation (Auszug):

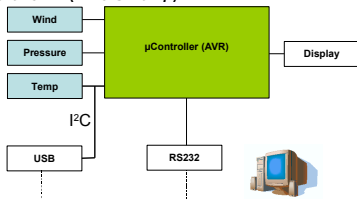
1. Sensordaten lesen
 - 1.1 Temperatursensor lesen
 - 1.2 Drucksensor lesen
 - 1.3 Windsensor lesen
2. Daten aufbereiten (z. B. glätten)
3. Daten ausgeben
4. Warten und ab Schritt 1 wiederholen



Funktionale Dekomposition: Beispiel

Funktionale Dekomposition der Wetterstation (Auszug):

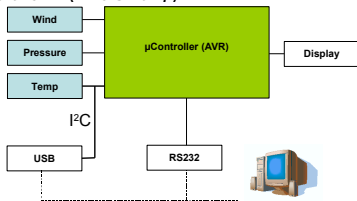
1. Sensordaten lesen
 - 1.1 Temperatursensor lesen
 - 1.1.1 I²C-Datenübertragung initiieren
 - 1.1.2 Daten vom I²C-Bus lesen
 - 1.2 Drucksensor lesen
 - 1.3 Windsensor lesen
2. Daten aufbereiten (z. B. glätten)
3. Daten ausgeben
4. Warten und ab Schritt 1 wiederholen



Funktionale Dekomposition: Beispiel

Funktionale Dekomposition der Wetterstation (Auszug):

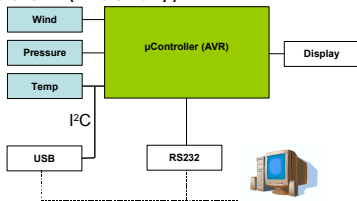
1. Sensordaten lesen
 - 1.1 Temperatursensor lesen
 - 1.1.1 I²C-Datenübertragung initiieren
 - 1.1.2 Daten vom I²C-Bus lesen
 - 1.2 Drucksensor lesen
 - 1.3 Windsensor lesen
2. Daten aufbereiten (z. B. glätten)
3. Daten ausgeben
 - 3.1 Daten über RS232 versenden
 - 3.2 LCD-Display aktualisieren
4. Warten und ab Schritt 1 wiederholen



Funktionale Dekomposition: Beispiel

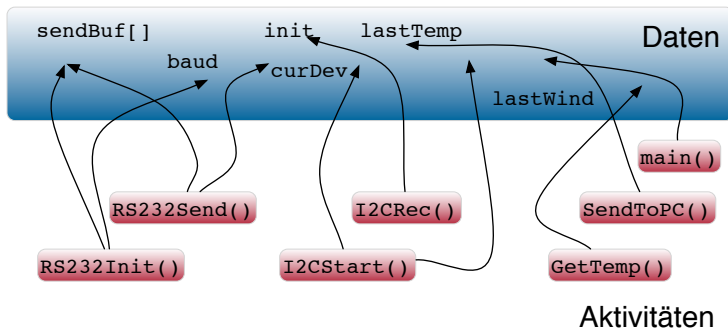
Funktionale Dekomposition der Wetterstation (Auszug):

1. Sensordaten lesen
 - 1.1 Temperatursensor lesen
 - 1.1.1 I²C-Datenübertragung initiieren
 - 1.1.2 Daten vom I²C-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 Struktur der **Daten**
- Gefahr: Funktionen arbeiten „wild“ auf einer Unmenge schlecht strukturierter Daten $\rightsquigarrow$ mangelhafte Trennung der Belange



- Erzielte Gliederung betrachtet nur die Struktur der **Aktivitäten**, nicht jedoch die Struktur der **Daten**
- Gefahr: Funktionen arbeiten „wild“ auf einer Unmenge schlecht strukturierter Daten $\rightsquigarrow$ 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).



Zugriff auf Daten (Variablen)

■ Variablen haben

↔ 10-1

- Sichtbarkeit (*Scope*) „Wer kann auf die Variable zugreifen?“
- Lebensdauer „Wie lange steht der Speicher zur Verfügung?“

■ Wird festgelegt durch Position (Pos) und Speicherklasse (SK)

Pos	SK	↔	Sichtbarkeit	Lebensdauer
Lokal	keine, auto static		Definition → Blockende Definition → Blockende	Definition → Blockende Programmstart → Programmende
Global	keine static		unbeschränkt modulweit	Programmstart → Programmende Programmstart → Programmende



Zugriff auf Daten (Variablen)

■ Variablen haben

↪ 10-1

- Sichtbarkeit (*Scope*) „Wer kann auf die Variable zugreifen?“
- Lebensdauer „Wie lange steht der Speicher zur Verfügung?“

■ Wird festgelegt durch Position (Pos) und Speicherklasse (SK)

Pos	SK	↪	Sichtbarkeit	Lebensdauer
Lokal	<i>keine</i> , auto static		Definition → Blockende Definition → Blockende	Definition → Blockende Programmstart → Programmende
Global	<i>keine</i> static		unbeschränkt modulweit	Programmstart → Programmende Programmstart → Programmende

```
int a = 0;           // a: global
static int b = 47;   // b: local to module

void f(void) {
    auto int a = b;   // a: local to function (auto optional)
                     // destroyed at end of block
    static int c = 11; // c: local to function, not destroyed
}
```



- Sichtbarkeit und Lebensdauer sollten **restriktiv** ausgelegt werden
 - Sichtbarkeit so **beschränkt wie möglich!**
 - Überraschende Zugriffe „von außen“ ausschließen (Fehlersuche)
 - Implementierungsdetails verbergen (Black-Box-Prinzip, *information hiding*)
 - Lebensdauer so **kurz wie möglich**
 - Speicherplatz sparen
 - Insbesondere wichtig auf μ -Controller-Plattformen

↪ 1-4



- Sichtbarkeit und Lebensdauer sollten **restriktiv** ausgelegt werden
 - Sichtbarkeit so **beschränkt wie möglich!**
 - Überraschende Zugriffe „von außen“ ausschließen (Fehlersuche)
 - Implementierungsdetails verbergen (Black-Box-Prinzip, *information hiding*)
 - Lebensdauer so **kurz wie möglich**
 - Speicherplatz sparen
 - Insbesondere wichtig auf μ -Controller-Plattformen

↪ 1-4

Konsequenz: Globale Variablen vermeiden!

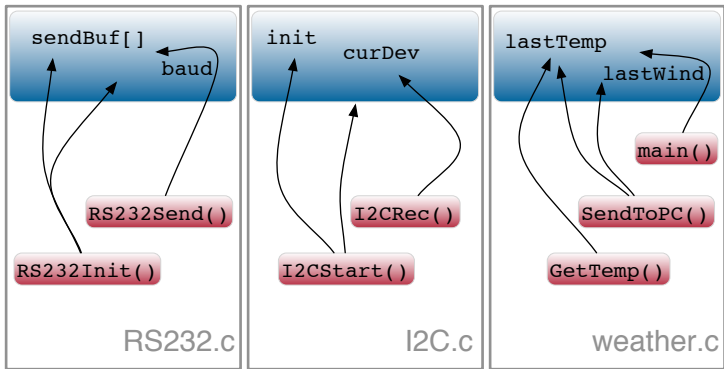
- Globale Variablen sind überall sichtbar
- Globale Variablen belegen Speicher über die gesamte Programmlaufzeit

Regel: Variablen erhalten stets die
geringstmögliche Sichtbarkeit und Lebensdauer



Lösung: Modularisierung

- Separation jeweils zusammengehöriger **Daten** und **Funktionen** in übergeordnete Einheiten $\rightsquigarrow$ **Module**



Was ist ein Modul?

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



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



- In C ist das Modulkonzept nicht Bestandteil der Sprache, ↪ 3-16
sondern rein **idiomatisch** (über **Konventionen**) realisiert
 - Modulschnittstelle ↪ .h-Datei (enthält Deklarationen ↪ 9-7)
 - Modulimplementierung ↪ .c-Datei (enthält Definitionen ↪ 9-3)
 - Modulverwendung ↪ `#include <Modul.h>`

```
extern void Init(uint16_t br);  
extern void Send(char ch);  
...
```

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



- In C ist das Modulkonzept nicht Bestandteil der Sprache, ↪ 3-16
sondern rein **idiomatisch** (über **Konventionen**) realisiert
 - Modulschnittstelle ↪ .h-Datei (enthält Deklarationen ↪ 9-7)
 - Modulimplementierung ↪ .c-Datei (enthält Definitionen ↪ 9-3)
 - Modulverwendung ↪ `#include <Modul.h>`

```
extern void Init(uint16_t br);  
extern void Send(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 Init(uint16_t br) {  
    ...  
    baud = br;  
}  
void Send(char ch) {  
    sendBuf[...] = ch;  
    ...  
}
```

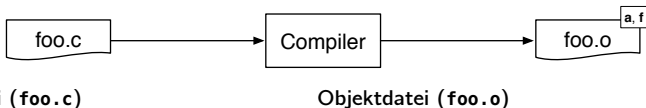
RS232.c: Implementierung (nicht öffentl.)
Definition der bereitgestellten Funktionen (und ggf. Daten)

Ggf. modulinterne Hilfsfunktionen und Daten (static)

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



- 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 ↔ 12-5)
- Export erfolgt beim Übersetzungsvorgang (.c-Datei → .o-Datei)



```
uint16_t a;  
// public  
static uint16_t b;  
// private  
  
void f(void)           // public  
{ ... }  
static void g(int)     // private  
{ ... }
```

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

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



- 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 uint16_t a;    // declare
extern void f(void);  // declare

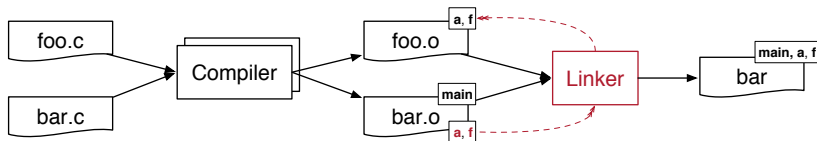
void main(void) {    // public
    a = 0x4711;       // use
    f();              // use
}
```

Objektdatei (**bar.o**)

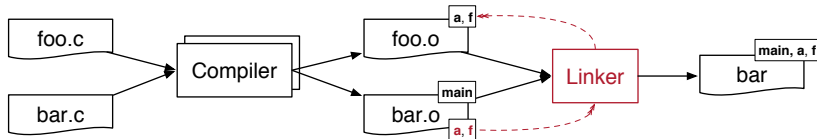
Symbol **main** wird exportiert.
Symbole **a** und **f** sind aufgelöst.



- Die eigentliche Auflösung erfolgt durch den **Linker**



- Die eigentliche Auflösung erfolgt durch den **Linker**



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

- Elemente aus fremden Modulen müssen deklariert werden

- Funktionen durch `extern` Deklaration

↪ 9-7

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

- Globale Variablen durch `extern`

```
extern uint16_t 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)



Modulschnittstelle: foo.h

```
// foo.h
#ifndef _F00_H
#define _F00_H

// declarations
extern uint16_t a;
extern void f(void);

#endif // _F00_H
```

Modulimplementierung foo.c

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

// definitions
uint16_t a;
void f(void) {
    ...
}
```

Modulverwendung bar.c

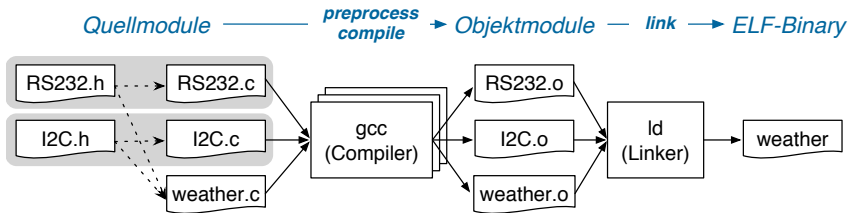
(vergleiche ↪ 12-11)

```
// bar.c
extern uint16_t a;
extern void f(void);
#include <foo.h>

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



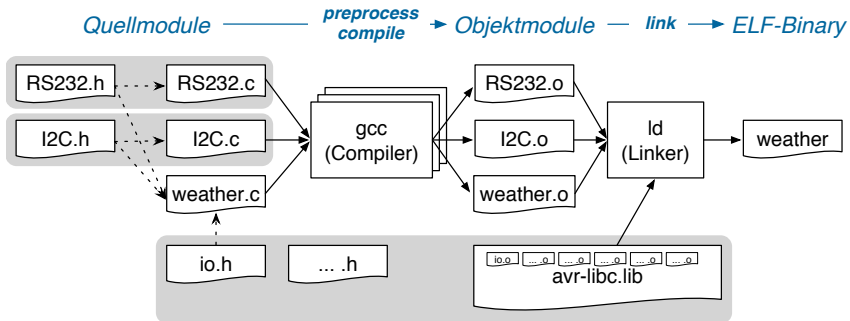
Zurück zum Beispiel: Wetterstation



- 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



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

- Prinzip der Trennung der Belange $\leadsto$ Modularisierung
 - Wiederverwendung und Austausch wohldefinierter Komponenten
 - Verbergen von Implementierungsdetails
- In C ist das Modulkonzept nicht Bestandteil der Sprache, sondern **idiomatisch** durch Konventionen realisiert
 - Modulschnittstelle $\mapsto$.h-Datei (enthält Deklarationen)
 - Modulimplementierung $\mapsto$.c-Datei (enthält Definitionen)
 - Modulverwendung $\mapsto$ `#include <Modul.h>`
 - **private** Symbole $\mapsto$ als `static` definieren
- Die eigentliche Zusammenfügung erfolgt durch den **Linker**
 - Auflösung erfolgt ausschließlich über Symbolnamen
 - $\leadsto$ **Linken ist nicht typsicher!**
 - Typsicherheit muss beim Übersetzen sichergestellt werden
 - $\leadsto$ durch gemeinsame Header-Datei

