System-Level Programming

12 Program Structure and Modules

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Summer Term 2025

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

Software Design

- Software design: general considerations about program's structure **before** the actual programming/implementation
  - Goal: Partitioning of the problem in manageable sub-problems

- There exists a multitude of different approaches for software design
  - Object-oriented approach
    - decomposition into classes and objects
    - designed for Python, Java, or C++
  - Top-down design / functional decomposition
    - state-of-the-art approach until the mid 80s
    - decomposition into functions and function calls
    - design constraints for FORTRAN, COBOL, Pascal, or C

Software Design

- Software design: general considerations about program's structure **before** the actual programming/implementation
  - Goal: Partitioning of the problem in manageable sub-problems

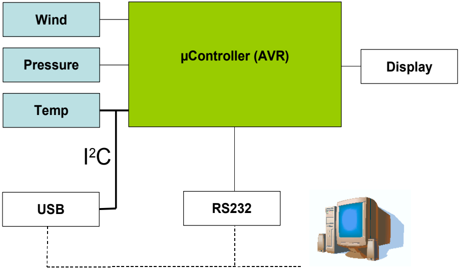
- There exists a multitude of different approaches for software design
  - Object-oriented approach
    - decomposition into classes and objects
    - designed for Python, Java, or C++
  - Top-down design / functional decomposition
    - state-of-the-art approach until the mid 80s
    - decomposition into functions and function calls
    - design constraints for FORTRAN, COBOL, Pascal, or C

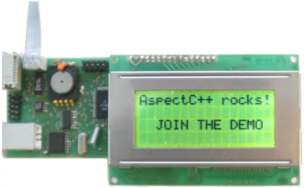
> System-level software is still designed with the **functional decomposition** in mind.

Example Project: A Weather Station

Typical embedded system

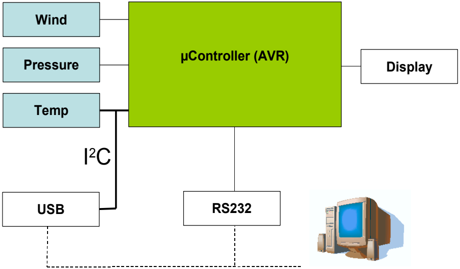
- multiple sensors
  - air speed
  - air pressure
  - temperature
- multiple actuators (here: output devices)
  - LCD-screen
  - PC via RS232
  - PC via USB
- Sensors and actuators are connected to the µC via different bus systems
  - I^2C
  - RS232



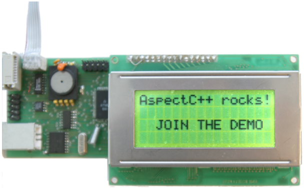


Example Project: A Weather Station

- Typical embedded system
  - multiple sensors
    - air speed
    - air pressure
    - temperature
  - multiple actuators (here: output devices)
    - LCD-screen
    - PC via RS232
    - PC via USB
  - Sensors and actuators are connected to the µC via different bus systems
    - I^2C
    - RS232



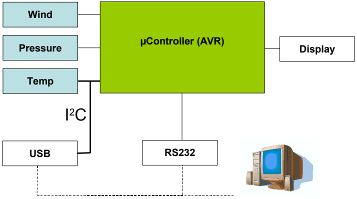
What does **functional decomposition** of the software look like?



Functional Decomposition: Example

Functional decomposition of the weather station:

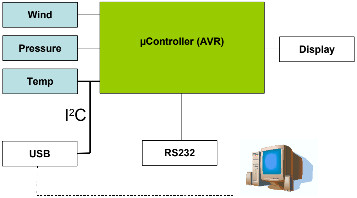
1. read sensor data
2. process data (e. g., smoothing)
3. output data
4. wait and eventually re-start again with step 1



Functional Decomposition: Example

Functional decomposition of the weather station:

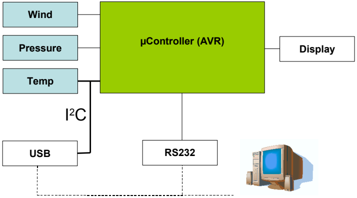
1. read sensor data
   - 1.1 read the temperature sensor
   - 1.2 read the pressure sensor
   - 1.3 read the air speed sensor
2. process data (e. g., smoothing)
3. output data
4. wait and eventually re-start again with step 1



Functional Decomposition: Example

Functional decomposition of the weather station:

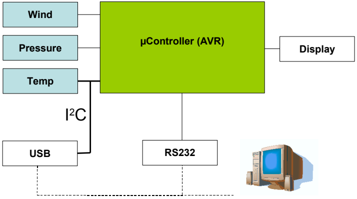
1. read sensor data
   1.1 read the temperature sensor
      1.1.1 initialize I²C data transfer
      1.1.2 read data from the I²C-bus
   1.2 read the pressure sensor
   1.3 read the air speed sensor
2. process data (e. g., smoothing)
3. output data
4. wait and eventually re-start again with step 1



Functional Decomposition: Example

Functional decomposition of the weather station:

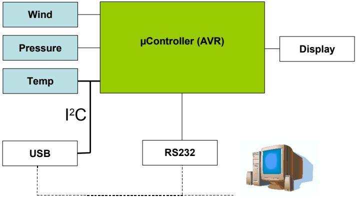
1. read sensor data
   - 1.1 read the temperature sensor
     - 1.1.1 initialize I^2C data transfer
     - 1.1.2 read data from the I^2C-bus
   - 1.2 read the pressure sensor
   - 1.3 read the air speed sensor
2. process data (e. g., smoothing)
3. output data
   - 3.1 sending data via RS232
   - 3.2 refresh the LCD
4. wait and eventually re-start again with step 1



Functional Decomposition: Example

Functional decomposition of the weather station:

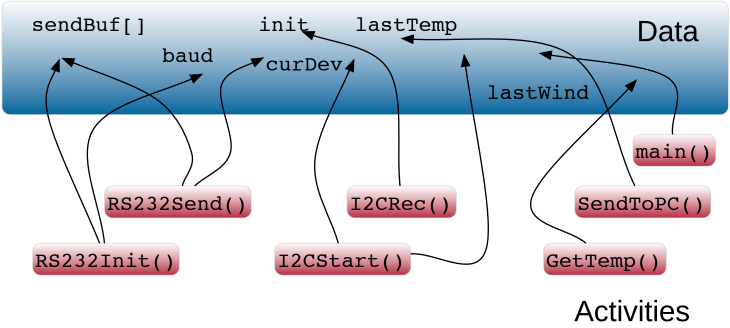
1. read sensor data
   - 1.1 read the temperature sensor
     - 1.1.1 initialize I²C data transfer
     - 1.1.2 read data from the I²C-bus
   - 1.2 read the pressure sensor
   - 1.3 read the air speed sensor
2. process data (e. g., smoothing)
3. output data
   - 3.1 sending data via RS232
     - 3.1.1 choose baud rate and parity (once)
     - 3.1.2 write data
   - 3.2 refresh the LCD
4. wait and eventually re-start again with step 1



Functional Decomposition: Problems

- The obtained decomposition does only account for the structure of the activities; however, not for the structure of the data
- Risk: Functions “wildly” work on a vast amount of unstructured data

$\rightsquigarrow$ inadequate separation of concerns



Functional Decomposition: Problems

- The obtained decomposition does only account for the structure of the activities; however, not for the structure of the data
- Risk: Functions “wildly” work on a vast amount of unstructured data
  $\rightsquigarrow$ inadequate separation of concerns

Principle of separation of concerns

Parts that have **nothing in common** with each other should be placed **separately**!

Separation of concerns is a **fundamental principle** in computer science (likewise in each other engineering discipline).

Access to Data (Variables)

- Variables have ↪ 10–1
  - Scope — "Who can access the variable?"
  - Lifespan — "How long is the memory accessible?"
- These are determined by their position (pos) and storage class (sc)

pos	sc ↦	scope	lifespan
local	*none*, `auto`	definition → end of block	definition → end of block
	`static`	definition → end of block	program start → program end
global	*none*	unrestricted	program start → program end
	`static`	whole module	program start → program end

Access to Data (Variables)

- Variables have ↪ 10–1
  - Scope — "Who can access the variable?"
  - Lifespan — "How long is the memory accessible?"
- These are determined by their position (pos) and storage class (sc)

pos	sc ↦	scope	lifespan
local	*none*, `auto`	definition → end of block	definition → end of block
	`static`	definition → end of block	program start → program end
global	*none*	unrestricted	program start → program end
	`static`	whole module	program start → program end

```c
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
}
```

Acces to Data (Variables) (continued)

- Scope and lifespan should be chosen restrictively
  - Scope as restricted as possible!
    - prevent unwanted access from other modules (debug)
    - hide information of implementation (black-box principle, *information hiding*)
  - Lifespan as short as possible!
    - save memory space
    - especially relevant for µController platforms ↪ 1–4

Acces to Data (Variables) (continued)

- Scope and lifespan should be chosen restrictively
  - Scope as restricted as possible!
    - prevent unwanted access from other modules (debug)
    - hide information of implementation (black-box principle, *information hiding*)
  - Lifespan as short as possible!
    - save memory space
    - especially relevant for µController platforms ↪ 1–4

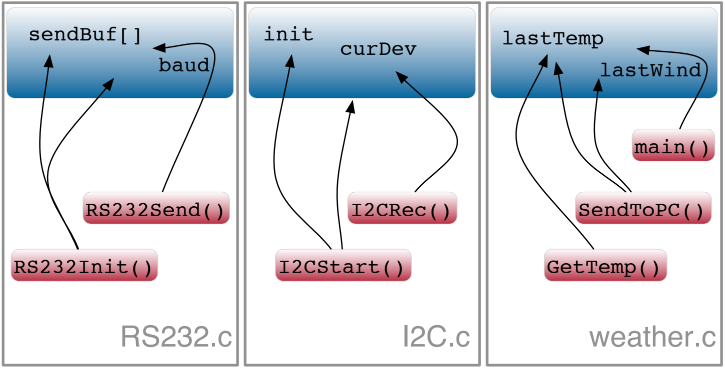
Consequence: Avoid global variables!

- global variables are visible everywhere
- global variables require memory for the entire program execution

Rule: Declaration of variables with **minimal scope & lifespan**

Solution: Modularisation

- Decomposition of related data & functions into dedicated, surrounding units ⇝ **modules**



What is a Module?

- **module** := (*<set of functions>*, *<set of data>*,
  <interface>)
- Modules are larger components of programs ↪ 9–1
  - problem-oriented aggregation of functions and data
    ⇝ separation of concerns
  - enable easy reuse of components
  - enable simple exchange of components
  - hide information of implementation: **black-box** principle
    ⇝ access only by means of the module's interface

What is a Module?

- **module** := (<*set of functions*>, <*set of data*>,
  <*interface*>)
- Modules are larger components of programs ↪ 9–1
  - problem-oriented aggregation of functions and data
    ⇝ separation of concerns
  - enable easy reuse of components
  - enable simple exchange of components
  - hide information of implementation: **black-box** principle
    ⇝ access only by means of the module's interface

Module ↦ Abstraction ↪ 4–1

- The interface of a module abstracts
  - from the actual implementation of the functions
  - from the internal representation and use of data

Modules in C [≠Python]

- In C, the modules are not part of the language itself, ↪ 3–15
  instead it is handled solely idiomatically (by using conventions)
  - module interface ↦ `.h`-file (contains declarations ↪ 9–7)
  - module implementation ↦ `.c`-file (contains definitions ↪ 9–3)
  - module usage ↦ `#include <module.h>`

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

RS232.h: Interface / Contract (public)
Declaration of provided functions (and data)

Modules in C [≠Python]

- In C, the modules are not part of the language itself, instead it is handled solely idiomatically (by using conventions) ↪ 3–15
  - module interface ↦ .h-file (contains declarations ↪ 9–7)
  - module implementation ↦ .c-file (contains definitions ↪ 9–3)
  - module usage ↦ `#include <module.h>`

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

RS232.h: Interface / Contract (public)
Declaration of provided functions (and data)

```c
#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;
  ...
}
```

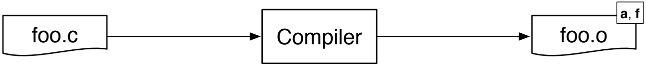
RS232.c: Implementation (not public)
Definition of provided functions (and data)

Possible module-internal helper functions and variables (`static`)

Inclusion of the own interface ensures that the contract is adhered to

Modules in C – Export [≠Python]

- C module exports a set of defined symbols
  - all functions and global variables
  - export can be prevented with `static` (↦ "__" **convention** in Python) (↦ restriction of scope ↪ 12–5)
- Export takes place during compilation (`.c` file ⟶ `.o` file)



source file (`foo.c`)

```
uint16_t a;          // public
static uint16_t b;   // private

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

object file (`foo.o`)

Symbols **a** and **f** are exported.

Symbols `b` and `g` are declared as `static` and, therefore, they are not exported.

Modules in C – Import

- C module imports a set of not-defined symbols
  - functions and global variables that are used but not defined in the module itself
  - during compilation, they are marked as unresolved

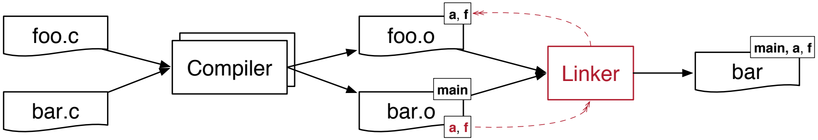
source file (bar.c)

```c
extern uint16_t a;   // declare
extern void f(void); // declare

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

object file (bar.o)

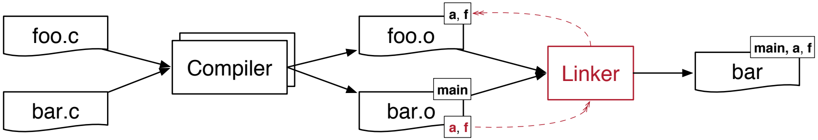
Symbol **main** is exported.
Symbols **a** and **f** are unresolved.

- The actual resolution is performed by the linker



Modules in C – Import (continued) [≠Python]

- The actual resolution is performed by the linker



Linking is **not type safe!**

- Information about types is not anymore present in the object files
- Resolution by the linker takes place **exclusively** via **names of symbols** (identifier)

⇝ type safety has to be ensured during **compilation**

⇝ uniform declaration with the help of a common header file

Modules in C – Header [≠Python]

- Elements from other modules have to be declared
  - functions with the `extern` declaration ↪ 9–7

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

  - global variables with `extern`

    ```
    extern uint16_t a;
    ```

> The keyword `extern` differentiates between a declaration and definition of a variable.

- Declarations are usually part of the header file, which module developers make available
  - interface of the module
    - exported functions of the module
    - exported global variables of the module
    - module-specific constants, types, and macros
    - usage by including (↦ "`import`" in Python)
  - is **included by the module itself** to ensure a match of declaration and definition

Modules in C – Header (continued) [≠Python]

module interface: `foo.h`

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

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

#endif // _FOO_H
```

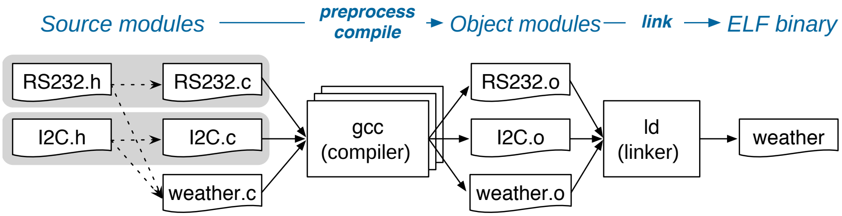
module implementation `foo.c`

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


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

module usage `bar.c`
(compare for ↪ 12–11)

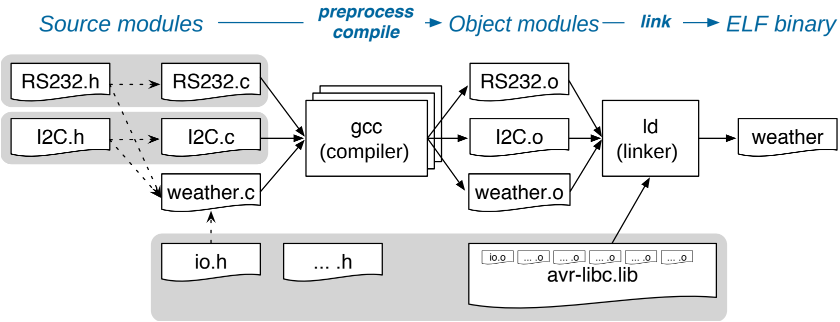
```c
// bar.c
~~extern uint16_t a;~~
~~extern void f(void);~~
#include <foo.h>

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

Back to the Example: Weather Station



- Each module consists of a header and one or more implementation file(s)
  - `.h` file **defines the interface**
  - `.c` file **implements the interface**, includes the `.h`-file to ensure a match of declaration and definition
- Usage of the module by including the specific `.h` file

Back to the Example: Weather Station



- Each module consists of a header and one or more implementation file(s)
  - `.h` file **defines the interface**
  - `.c` file **implements the interface**, includes the `.h`-file to ensure a match of declaration and definition
- Usage of the module by including the specific `.h` file
- This is similar for libraries

Summary

- Principle of separation of concerns ⇝ modularization
  - reuse and exchange of well-defined components
  - hiding of implementation details
- In C, the concept of modules is not part of the language. Therefore, it is realized idiomatically by conventions.
  - module interface ↦ `.h`-file (contains declarations)
  - module implementation ↦ `.c`-file (contains definitions)
  - use of module ↦ `#include <module.h>`
  - private symbols ↦ define as `static`
- The actual combination is done by the linker
  - resolution exclusively by symbol names
    ⇝ **Linking is not type safe!**
  - type safety has to be ensured during compilation
    ⇝ with the help of a common header file