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 2024

http://sys.cs.fau.de/lehre/ss24

Software Design

- Software design: general considerations about structure of a program **before** the actual programming/implementation starts
  - 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 Java or C++
  - Top-down design/functional decomposition
    - state of the art until the mid 80s
    - decomposition into functions and function calls
    - design constraints for FORTRAN, COBOL, Pascal, or C

Software Design

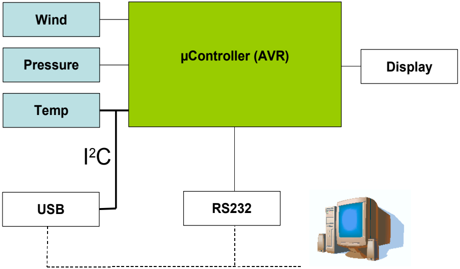
- Software design: general considerations about structure of a program **before** the actual programming/implementation starts
  - Goal: Partitioning of the problem in manageable sub-problems

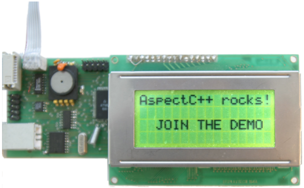
- There exists a multitude of different approaches for software design
  - Object-oriented approach
    - decomposition into classes and objects
    - designed for Java or C++
  - Top-down design/functional decomposition
    - state of the art until the mid 80s
    - decomposition into functions and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

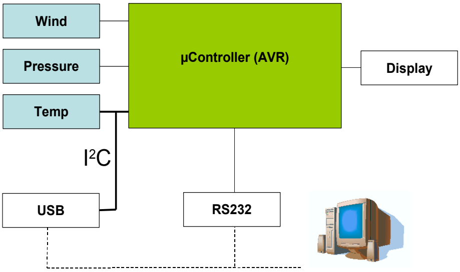
- Typical embedded system
  - multiple sensors
    - air speed
    - air pressure
    - temperature
  - multiple actuators (here: output devices)
    - LCD-screen
    - PC via RS232
    - PC via USB
  - Sensors and actuators are connected to the μC via different bus systems
    - I^2C
    - RS232



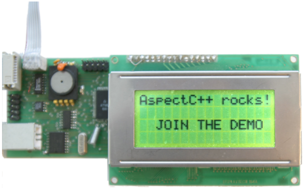


Example Project: A Weather Station

- 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?

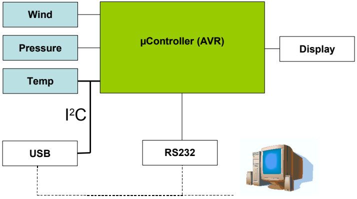


12-Module_en

Functional Decomposition: Example

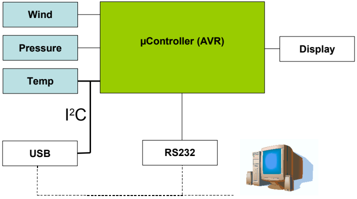
Functional decomposition of the weather station (extract):

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

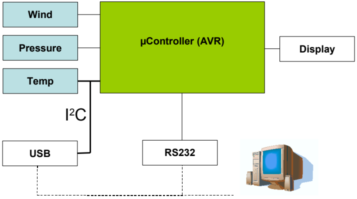
Functional decomposition of the weather station (extract):



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

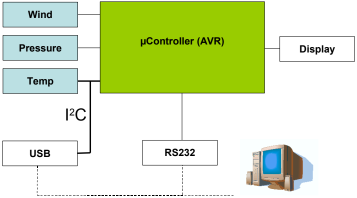
Functional decomposition of the weather station (extract):



1. read sensor data
   1.1 read the temperature sensor
      1.1.1 initialize I^{2}C data transfer
      1.1.2 read data from the I^{2}C-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

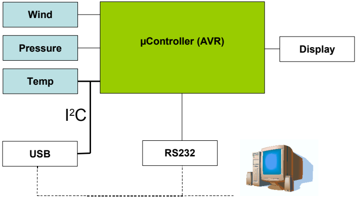
Functional decomposition of the weather station (extract):



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 (extract):

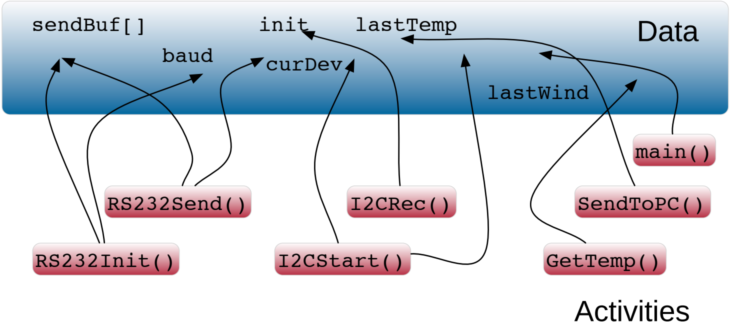


1. read sensor data
   - 1.1 read the temperature sensor
     - 1.1.1 initialize I²C data transfer
     - 1.1.2 read data from the I²C-bus
   - 1.2 read the pressure sensor
   - 1.3 read the air speed sensor
2. process data (e. g., smoothing)
3. output data
   - 3.1 sending data via RS232
     - 3.1.1 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

$\leadsto$ 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 ↪ ??
  - Scope "Who can access the variable?"
  - Lifespan "How long is the memory accessible?"
- These get set by 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 ↪ ??
  - Scope — "Who can access the variable?"
  - Lifespan — "How long is the memory accessible?"
- These get set by 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 the implementation (black-box principle)
  - Lifespan as short as possible!
    - save memory space
    - especially relevant for μController platforms ↪ ??

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 the implementation (black-box principle)
  - Lifespan as short as possible!
    - save memory space
    - especially relevant for μController platforms ↪ ??

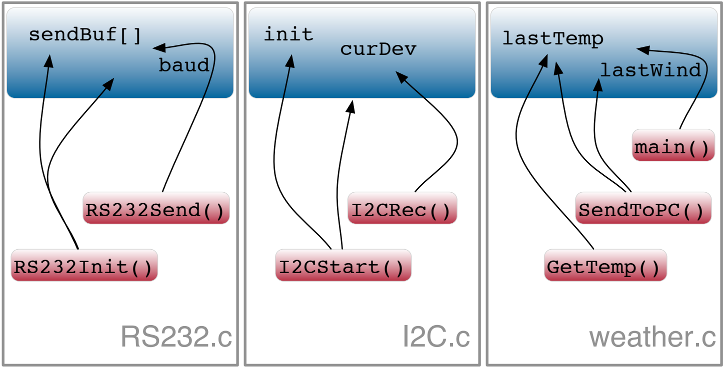
Consequence: Avoid global variables!

- 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**



12-Module_en

What is a Module?

- **module** := (<*set of functions*>, (↦ "`class`" in Java)
  <*set of data*>,
  <*interface*>)
- Modules are larger programming components ↪ 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*>, (↦ "`class`" in Java)
  <*set of data*>,
  <*interface*>)
- Modules are larger programming components ↪ 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 ↪ **??**

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

- In C, the modules are not part of the language itself, ↪ ??
  instead it is handled solely idiomatically (with help of conventions)
  - module interface ↦ .h-file (contains declarations ↪ 9–9)
  - module implementation ↦ .c-file (contains definitions ↪ 9–4)
  - module usage ↦ `#include <module.h>`

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

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

12-Module_en

Modules in C [≠Java]

- In C, the modules are not part of the language itself, ↪ ??
  instead it is handled solely idiomatically (with help of conventions)
  - module interface ↦ .h-file (contains declarations ↪ 9–9)
  - module implementation ↦ .c-file (contains definitions ↪ 9–4)
  - module usage ↦ `#include <module.h>`

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

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

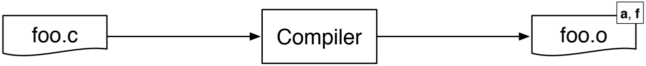
```
#include <RS232.h>
static uint16_t   baud = 2400;
static char       sendBuf[16];
...
void Init(uint16_t br) {
  ...
  baud = br;
}
void Send(char ch) {
  sendBuf[···] = ch;
  ...
}
```

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

- C module exports a set of defined symbols
  - all functions and global variables (↦ "`public`" in Java)
  - export can be prevented with `static` (↦ "`private`" in Java)
    (↦ 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 [≠Java]

- 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`)

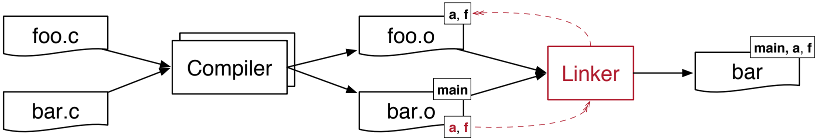
```
extern uint16_t a;   // declare
extern void f(void); // declare

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

object file (`bar.o`)

Symbol **`main`** is exported.
Symbols **`a`** and **`f`** are unresolvedt.

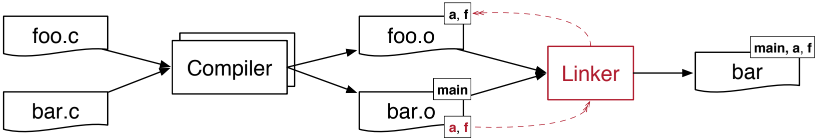
Modules in C – Import (continued) [≠Java]

- The actual resolution is performed by the linker



Modules in C – Import (continued)

- 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 [≠Java]

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

    ```
    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 take place usually in a header file, which is made available by the module developers
  - interface of the module (↦ “interface” in Java)
    - exported functions of the module
    - exported global variables of the module
    - module specific constants, types and macros
    - usage by inclusion (↦ “import” in Java)
  - is **included by the module itself** to ensure a match of declaration and definition (↦ “implements” in Java)

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

module interface: `foo.h`

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

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

#endif // _FOO_H
```

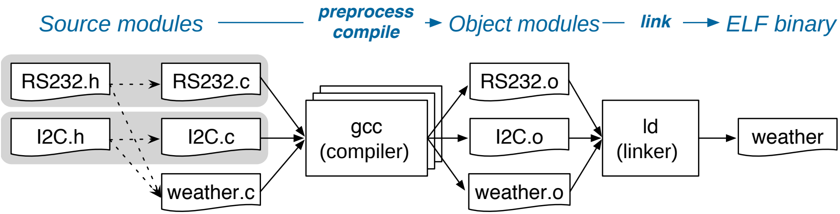
module implementation `foo.c`

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


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

module uasge `bar.c`
(compare to ↪ 12–13)

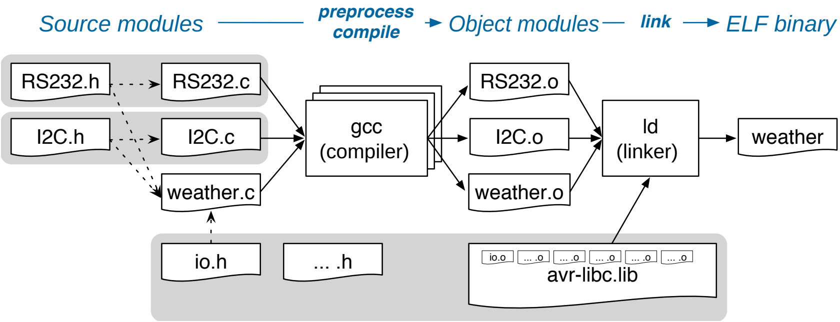
```
// bar.c
```
~~`extern uint16_t a;`~~
~~`extern void f(void);`~~
```
#include <foo.h>

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
- Use 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
- Use of the module by including the specific `.h`-file
- This is similar for libraries

Summary

- Principle of separation of concerns $\rightsquigarrow$ modularisation
  - 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 has to be made possible idiomatically by conventions
  - module interface $\mapsto$ `.h`-file (contains declarations)
  - module implementation $\mapsto$ `.c`-file (contains definitions)
  - use of module $\mapsto$ `#include <module.h>`
  - `private` symbols $\mapsto$ define as `static`
- The actual combination is done by the linker
  - resolution exclusively by symbol names
    $\rightsquigarrow$ **Linking is not type safe!**
  - type safety has to be ensured during compilation
    $\rightsquigarrow$ with the help of a common header file