

# Verlässliche Echtzeitsysteme

## Übungen zur Vorlesung

Tobias Klaus, Florian Franzmann

Friedrich-Alexander-Universität Erlangen-Nürnberg  
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)  
<https://www4.cs.fau.de>

28. April 2015



- C99
- x86 bzw. x86-64, d. h.
  - vorzeichenbehaftete Integer als Zweierkomplement implementiert
  - char hat 8 Bit
  - short hat 16 Bit
  - int hat 32 Bit
  - long hat 32 Bit auf x86 und 64 Bit auf x86-64



## Frage 4

Zu was wird  $-1L > 1U$  auf x86-64 ausgewertet? Auf x86?

1. beides 0
2. beides 1
3. 0 auf x86-64, 1 auf x86
4. 1 auf x86-64, 0 auf x86

### Erklärung

- auf x86-64 ist **int** kürzer als **long**
- ↪ **unsigned int** wird zu **long** ↪  $-1L > 1L \Rightarrow 0$
- auf x86 entspricht **int** dem Datentyp **long**
- ↪  $UINT\_MAX > 1U \Rightarrow 1$



## Frage 5

Zu was wird `SCHAR_MAX == CHAR_MAX` ausgewertet?

1. 0
2. 1
3. nicht definiert

### Erklärung

- C99 schreibt nicht vor ob `char` vorzeichenbehaftet ist
- auf x86 und x86-64 ist `char` für gewöhnlich vorzeichenbehaftet



## Frage 6

Zu was wird `UINT_MAX + 1` ausgewertet?

1. `0`
2. `1`
3. `INT_MIN`
4. `UINT_MIN`
5. nicht definiert

### Erklärung

Der C-Standard garantiert, dass `UINT_MAX + 1 == 0`



**Modifizierbarkeit:** lokale Veränderbarkeit

- ~> Änderungen an Anforderungen umsetzbar
- ~> Fehler korrigierbar

**Effizienz:** optimaler Betriebsmittelbedarf

- wird häufig zu früh berücksichtigt

**Verlässlichkeit:** über lange Zeit Funktionsfähigkeit ohne menschlichen Eingriff

- gutmütiges Ausfallverhalten
- muss von Anfang an eingeplant sein!

**Verständlichkeit:** Isolierung von

- Daten
- Algorithmen



# Prinzipien des Softwareentwurfs

---

**Abstraktion:** wichtige Details hervorheben

**Kapselung:** unnötige Details verbergen

**Einheitlichkeit:** konsistente Notation

**Vollständigkeit:** alle wichtigen Aspekte berücksichtigt

**Testbarkeit:** muss von Anfang an eingeplant werden

**C macht es einem hier nicht leicht**

~> disziplinierte Herangehensweise notwendig!



*Wie komme ich von der Beschreibung zur Software?*

## Objektorientierter/Objektbasierter Entwurf [?]

1. identifiziere Objekte und deren Attribute
2. identifiziere Operationen jedes Objekts
3. lege Sichtbarkeit fest
4. lege Objektschnittstellen fest
5. implementiere Objekte





## Beispielbeschreibung – $\alpha$ -Filter

- $\hat{x}[\kappa]$  Schätzung,  $\alpha$  Filterparameter,  $y[\kappa]$  Messwert
- Initialisierung:  $\hat{x}[0] = 0$
- Filterschritt:

$$r[\kappa] = y[\kappa] - \hat{x}[\kappa - 1] \quad (1)$$

$$\hat{x}[\kappa] = \hat{x}[\kappa - 1] + \alpha \cdot r[\kappa] \quad (2)$$

- Optimale Parameter ( $\sigma_w^2$  Prozessvarianz,  $T$  Abtastintervall,  $\sigma_v^2$  Rauschvarianz):

$$\lambda = \sigma_w \cdot T^2 / \sigma_v \quad (3)$$

$$\alpha = \frac{-\lambda^2 + \sqrt{\lambda^4 + 16\lambda^2}}{8} \quad (4)$$



# 1. Objekte und Attribute identifizieren

- Herangehensweise:
  - Hauptwortextraktion aus Anforderungsdokument
  - für kleinere Probleme: *Intuition*
- Was ist das Objekt?  $\rightsquigarrow$  Filter
- Attribute? Welche Information brauche ich für jeden Filterschritt?
  - Schätzung aus der Vorrunde  $\hat{x}[\kappa - 1]$
  - Filterparameter  $\alpha$
  - aktuellen Messwert  $y[\kappa] \rightsquigarrow$  kein Zustand, kommt von aussen

## Vorläufige Objektschablone

```
1 typedef struct _Alpha_Filter {
2     AF_Value_t x;
3     AF_Value_t alpha;
4 } Alpha_Filter;
```



## 2. Operationen identifizieren

---

- Herangehensweise:
  - Verbenextraktion
  - für kleinere Probleme: *Intuition*
- Leben eines Objekts:
  1. Initialisierung  $\leadsto$  Betriebsmittel anfordern
  2. Verwendung
  3. Beseitigung  $\leadsto$  Betriebsmittel freigeben
- Was möchten Benutzer mit dem Filter machen?
  - Filter initialisieren
  - Filterschritt ausführen
  - Schätzwert erfragen
  - Betriebsmittelfreigabe nicht notwendig



### 3. Sichtbarkeit festlegen

---

- in modernen Programmiersprachen `private`, `public`, ...
- in C nur eingeschränkt möglich
  - `modulintern` vs. `modulextern`
- Leitfaden: möglichst wenig sichtbar machen
  - ↳ öffentliche Schnittstelle bedeutet Verpflichtung
- Was soll bei unserem Filter öffentlich sein?
  - Initialisierung
  - Filterschritt
  - Schätzung abfragen
- alle anderen Operationen `modulintern`
  - ↳ Hilfsfunktionen `static`



## 4. Schnittstelle festlegen

- zwischen Modul und Außenwelt
- statische Semantik

### Schnittstelle

```
1 void afilter_init(Alpha_Filter *filter,  
2                 AF_Value_t process_variance,  
3                 AF_Value_t noise_variance,  
4                 AF_Value_t sampling_interval);  
5  
6 void afilter_step(Alpha_Filter *filter,  
7                 AF_Value_t measurement);  
8  
9 AF_Value_t afilter_get_estimate(Alpha_Filter *filter);
```



## 5. Implementierung – Header

### alpha\_filter.h

```
1 #ifndef ALPHA_FILTER_H_INCLUDED
2 #define ALPHA_FILTER_H_INCLUDED
3
4 typedef float AF_Value_t;
5 typedef struct _Alpha_Filter {
6     AF_Value_t x;
7     AF_Value_t alpha;
8 } Alpha_Filter;
9
10 void afilter_init(Alpha_Filter *filter,
11                 AF_Value_t process_variance,
12                 AF_Value_t noise_variance,
13                 AF_Value_t sampling_interval);
14
15 void afilter_step(Alpha_Filter *filter,
16                 AF_Value_t measurement);
17
18 AF_Value_t afilter_get_estimate(Alpha_Filter *filter);
19 #endif // ALPHA_FILTER_H_INCLUDED
```



## 5. Implementierung – Initialisierung

$$\hat{x}[0] = 0 \quad (5)$$

$$\lambda = \sigma_w \cdot T^2 / \sigma_v \quad (6)$$

$$\alpha = \left( -\lambda^2 + \sqrt{\lambda^4 + 16\lambda^2} \right) / 8 \quad (7)$$

### alpha\_filter.c

```
1 void afilter_init(Alpha_Filter *filter,  
2                 AF_Value_t process_variance,  
3                 AF_Value_t noise_variance,  
4                 AF_Value_t sampling_interval) {  
5     filter->x = 0;  
6     AF_Value_t l = sqrt(process_variance)  
7         * sampling_interval * sampling_interval  
8         / sqrt(noise_variance);  
9     filter->alpha = (-l * l  
10        + sqrtf(l * l * l * l + 16.0f * l * l)) / 8.0f;  
11 }
```

## 5. Implementierung – Filterschritt

$$r[\kappa] = y[\kappa] - \hat{x}[\kappa - 1] \quad (8)$$

$$\hat{x}[\kappa] = \hat{x}[\kappa - 1] + \alpha \cdot r[\kappa] \quad (9)$$

### alpha\_filter.c

```
1 void afilter_step(Alpha_Filter *filter,
2                 AF_Value_t measurement) {
3     AF_Value_t r = measurement - filter->x;
4     filter->x = filter->x + filter->alpha * r;
5 }
6
7 AF_Value_t afilter_get_estimate(Alpha_Filter *filter) {
8     return filter->x;
9 }
```





# Zwei Prinzipien für die Übung

---

*KISS* – Keep it Small and Simple!

- Kleine Softwaremodule mit geringer Kopplung
  - *Eine* (C-)Funktion löst *eine* Aufgabe
  - ☞ Bessere Wartbarkeit, Testbarkeit, Verifizierbarkeit
- 
- Ein Beispiel: libmathe16



## *DRY* – Don't repeat yourself!

- Code nicht unnötig duplizieren
  - Oft benutzten (getesteten) Code wiederverwenden
  - ☞ Einsatz von Bibliotheken
  - ☞ Befehle (gcc/ar/...) nicht unnötig händisch wiederholen
- 
- Stupidies Wiederholen von Befehlen ist fehlerträchtig!
  - Lösung: Buildsystem
    - Automatisiertes Bauen
    - Automatisches Auflösen von Abhängigkeiten
    - Viele existierende Lösungen: make, ANT, Maven, u.v.m.
  - Wir nutzen *CMake*



- 1 C-Quiz Teil II
- 2 Softwareentwurf
- 3 Zuverlässig Software entwickeln
- 4 CMake – Ein Meta-Buildsystem**
- 5 gdb
- 6 Übungsaufgabe



- Ein Meta-Buildsystem!
  - Erzeugt Buildsystemdateien
    - *Makefiles* (GNU, NMake, ...)
    - *ninja*
    - Projektdateien (KDevelop, Eclipse, Visual Studio, Xcode)
  - Einfache, skriptähnliche Sprache
  - Plattform-/Betriebssystemunabhängig
  - Ermöglicht „Out-of-Source Builds“
- Weit verbreitet
  - KDE, MySQL, LLVM, u.v.m.



- Konfigurationsdatei(en): CMakeLists.txt
- Separat in jedem Unterverzeichnis
  - Ausgehend vom Basisverzeichnis → `add_subdirectory(...)`
- Definition von sog. „Targets“
  - `add_executable(<Targetname> <Quelldatei1.c> <Quelldatei2.c>)`
  - `add_library(<Libraryname> <Quelldatei1.c> <Quelldatei2.c>)`
- Hinzubinden von Bibliotheken
  - `target_link_libraries(<Targetname> <Libraryname>)`
  - Abhängigkeiten werden automatisch erkannt
- Manuelle Festlegung von Abhängigkeiten
  - `add_dependency(<Targetname1> <Targetname2>)`



# Verzeichnisstruktur in der Übung

- Quellverzeichnis (source)
- Hier liegen die Quelldateien
  - include ← Schnittstellenbeschreibungen (.h)
  - src ← Implementierung (.c)
  - tests ← Testfallimplementierungen (.c)
  - (cmake) ← (eigene CMake Erweiterungen)
- Binärverzeichnis (build)
- Hier landen ausschließlich(!) generierte Dateien
  - Objektdateien (.o)
  - Bibliotheken (.a)
  - Ausführbare Dateien
- ☞ „Out-of-Source Build“
- ☞ Beispiel



- *Außerhalb* des Quellverzeichnisses
- % cmake <Pfad zum Quellverzeichnis>
- % make **help** zeigt alle möglichen Targets
- % cmake <Buildverzeichnis> zum Einstellen von Buildparametern

 Beispiel



- unterstützt die Integration von Tests im Softwareprojekt
- Automatisierte Ausführung und Auswertung von Testläufen
- Konfigurationsdatei: `tests/CMakeLists.txt`
  - Ausführbares Target:  
`add_executable(plus_test plus_test.c)`
  - Hinzubinden der zu testenden Bibliothek:  
`target_link_libraries(plus_test mathe)`
  - Bekanntmachen als Testfall:  
`add_test(MatheTest_PLUS plus_test)`
- `make test` führt Tests aus
- Automatische Testauswertung:
  - Anhand Rückgabewert (0 → OK, -1 → Fehler)
  - Notfalls auch Parsen von Ausgaben





- Werkzeug aus der gcc-Toolchain
- Instrumentierung des Binär-codes
- Protokollieren der Programmausführung
  - Wie oft wird jede Codezeile ausgeführt?
  - Welche Zeilen werden überhaupt ausgeführt?
  - Welche Verzweigungen wurden genommen?
- HTML Ausgabe: lcov
  - Tests solange verfeinern, bis alles überdeckt ist!
  - Ausführen automatisiert über `make lcov`



# Aufdecken von Laufzeitfehlern

- In dieser Übung: clang Sanitizer
  - Wird von unseren cmake-Skripten automatisch verwendet, wenn
    - Debugging aktiviert ist
    - und clang als Compiler verwendet wird
    - siehe cmake/sanitizer.cmake
  - *im CIP-Pool erst addpackage clang ausführen*
- ↪ CC=clang CXX=clang++ cmake -DCMAKE\_BUILD\_TYPE=Debug ..
- entdeckt z. B.
    - falsche Verwendung von Zeigern
    - nicht-definierte Integer-Operationen
    - lesen nichtinitialisierten Speichers
    - Integer-Überlauf

## Entdeckt Fehler ...

... nur, wenn die verwendeten Testfälle diese auslösen.



## Beispiel

## ■ Quellverzeichnis

```
% tree ~/source
~/source
|-- CMakeLists.txt
|-- include
|   '-- mathe.h
|-- src
|   |-- CMakeLists.txt
|   |-- abs.c
|   '-- plusminus.c
'-- tests
    |-- CMakeLists.txt
    |-- abs_test.c
    '-- plus_test.c
```

## ■ Binärverzeichnis

```
% cd ~/binary
% cmake ..
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Checking whether C compiler has -isysroot
...
-- Configuring done
-- Generating done
-- Build files have been written to: ~/build
% make
[ 20%] Building C object src/CMakeFiles/mathe.dir/plusminus.c.o
[ 40%] Building C object src/CMakeFiles/mathe.dir/abs.c.o
Linking C static library libmathe.a
[ 60%] Built target mathe
Scanning dependencies of target abs_test
[ 80%] Building C object tests/CMakeFiles/abs_test.dir/abs_test.c.o
Linking C executable abs_test
[ 80%] Built target abs_test
Scanning dependencies of target plus_test
[100%] Building C object tests/CMakeFiles/plus_test.dir/plus_test.c.o
Linking C executable plus_test
[100%] Built target plus_test
% make test
Running tests...
Test project ~/build
  Start 1: MatheTest_PLUS
1/2 Test #1: MatheTest_PLUS ..... Passed    0.00 sec
  Start 2: MatheTest_ABS
2/2 Test #2: MatheTest_ABS .....***Failed    0.00 sec
50% tests passed, 1 tests failed out of 2
Total Test time (real) =  0.02 sec

The following tests FAILED:
   2 - MatheTest_ABS (Failed)
Errors while running CTest
```



- 1 C-Quiz Teil II
- 2 Softwareentwurf
- 3 Zuverlässig Software entwickeln
- 4 CMake – Ein Meta-Buildsystem
- 5 gdb**
- 6 Übungsaufgabe



- Ein Debugger dient zum Suchen und Finden von Fehlern in Programmen
- Im Debugger kann man u.a.
  - das Programm schrittweise abarbeiten
  - Variablen- und Speicherinhalte ansehen und modifizieren
  - core dumps (Speicherabbilder beim Programmabsturz) analysieren
    - Erlauben von core dumps (in der laufenden Shell): z. B. `limit coredumpsize 1024k` oder `limit coredumpsize unlimited`
- Programm sollte Debug-Symbole enthalten
  - `ccmake .` → `CMAKE_BUILD_TYPE: Debug`
- Aufruf des Debuggers mit `gdb <Programmname>`
- in der Übungsaufgabe: `make cgdb`, `make gdbtui`



1. Objektbasierter Softwareentwurf
  2. Testfallentwurf
    - Vollständige Pfadeüberdeckung
    - Abdecken aller Randfälle
  3. Implementierung von Software und Testfällen
    - Getrennte Implementierung von Software und Testfällen
    - Möglichst durch verschiedene Übungsteilnehmer
- Implementiert werden soll eine *Prioritätswarteschlange*
  - einfügen, entfernen, *iterieren*
- ↪ `for (... x = ...; x != ...; ++x){...}`
- Implementierung?



- Datenstruktur als Array im Header vereinbaren
- Zugriff durch Zeigerarithmetik

```
1 typedef struct Element { ... } Element;  
2 Element elements[ELEMENTS_SIZE];  
3 ...  
4     for (size_t i = 0; i < ELEMENTS_SIZE; ++i)  
5         { use(elements[i]); }
```

- *Vorteile:*
  - Einfache Implementierung
  - Für den Compiler leicht zu optimieren
- *Nachteil:* Implementierung offengelegt  
~> Verpflichtung ggü. Benutzer



## ■ Iterator als Teil des Objekts

### ■ Header:

```
1 typedef struct Elements Elements;
2 void El_reset_iterator(Elements *self);
3 void El_next(Elements *self);
4 bool El_isAtEnd(Elements *self);
5 int64_t El_iterator_value(Elements *self);
```

### ■ Verwendung:

```
1 ...
2 El_reset_iterator(dings);
3 while(!El_isAtEnd(dings)) {
4     use(El_iterator_value(dings));
5     El_next(dings);
6 }
```





## ■ Implementierung:

```
1 typedef struct Element { int64_t value; } Element;
2 struct Elements {
3     Element elements[ELEMENTS_SIZE];
4     Element *it;
5 };
6 void El_reset_iterator(Elements *self)
7     { self->it = &self->elements }
8 void El_next(Elements *self)
9     { self->it = self->it + 1; }
10 bool El_isAtEnd(Elements *self)
11     { return self->it
12         == &(self->elements[ELEMENTS_SIZE]); }
13 int64_t El_iterator_value(Elements *self)
14     { return self->it->value; }
```

## ■ *Vorteil:* Kapselung sehr gut

## ■ *Nachteile:*

- Für den Compiler evt. nicht mehr optimierbar
- So nur ein Iterator gleichzeitig möglich



## ■ Iterator als eigenes Objekt

### ■ Header:

```
1 typedef struct Elements Elements;
2 typedef struct El_Iterator El_Iterator;
3
4 El_Iterator *El_begin(Elements *self);
5 void El_Iterator_destroy(El_Iterator *self);
6 void El_Iterator_next(El_Iterator *self);
7 bool El_Iterator_isAtEnd(El_Iterator *self);
8 int64_t El_Iterator_value(El_Iterator *self);
```

### ■ Verwendung:

```
1 ...
2 El_Iterator *it;
3 for (it = El_begin(dings);
4      not El_Iterator_isAtEnd(it);
5      El_Iterator_next(it)) {
6     use(El_Iterator_value(it))
7 }
8 El_Iterator_destroy(it);
9
```



## ■ Implementierung:

```
1 typedef struct Element { int64_t value; } Element;
2 struct Elements { Element elements[ELEMENTS_SIZE]; };
3 struct El_Iterator {
4     Element *position;
5     Element *end;
6 };
7 El_Iterator *El_begin(Elements *self) {
8     El_Iterator *ret = malloc(sizeof(El_Iterator));
9     if (ret == NULL) { return NULL; }
10    ret->position = self->elements;
11    ret->end = &self->elements[ELEMENTS_SIZE];
12    return ret;
13 }
14 void El_Iterator_next(El_Iterator *self)
15     { self->position += 1; }
16 bool El_Iterator_isAtEnd(El_Iterator *self) { ... }
17 int64_t El_Iterator_value(El_Iterator *self) { ... }
18 void El_Iterator_destroy(El_Iterator *self) { ... }
```



- *Vorteile:*

- Vollständige Kapselung
- Beliebig viele Iteratoren möglich

- *Nachteil:*

- Iterator muss nach Gebrauch beseitigt werden
- Compiler hat evt. Probleme zu optimieren



- Programmausführung beeinflussen
  - Breakpoints setzen:
    - b [<Dateiname>:]<Funktionsname>
    - b <Dateiname>:<Zeilennummer>
  - Starten des Programms mit run (+ evtl. Befehlszeilenparameter)
  - Fortsetzen der Ausführung bis zum nächsten Stop mit c (continue)
  - schrittweise Abarbeitung auf Ebene der Quellsprache mit
    - s (step: läuft in Funktionen hinein)
    - n (next: behandelt Funktionsaufrufe als einzelne Anweisung)
  - Breakpoints anzeigen: info breakpoints
  - Breakpoint löschen: delete breakpoint#



- Variableninhalte anzeigen/modifizieren
  - Anzeigen von Variablen mit: `p expr`
    - `expr` ist ein C-Ausdruck, im einfachsten Fall der Name einer Variable
  - Automatische Anzeige von Variablen bei jedem Programmstopp (Breakpoint, Step, ...): `display expr`
  - Setzen von Variablenwerten mit `set <variablenname>=<wert>`
- Ausgabe des Funktionsaufruf-Stacks (backtrace): `bt`
- Quellcode an aktueller Position anzeigen: `list`
- Watchpoints: Stoppt Ausführung bei Zugriff auf eine bestimmte Variable
  - `watch expr`: Stoppt, wenn sich der Wert des C-Ausdrucks `expr` ändert
  - `rwatch expr`: Stoppt, wenn `expr` gelesen wird
  - `awatch expr`: Stopp bei jedem Zugriff (kombiniert `watch` und `rwatch`)
  - Anzeigen und Löschen analog zu den Breakpoints



---

# Fragen?

