

Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Filter

Florian Schmaus, Simon Schuster

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

16. April 2018



- 1 C-Quiz Teil I
- 2 Festkommaarithmetik
- 3 Softwareentwurf
- 4 Hinweise zur Aufgabe: Filter



- 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 3

Angenommen: `int x = 1`; Zu was wird `(unsigned short)x > -1` ausgewertet?

1. 0
2. 1
3. nicht definiert

Erklärung

- vor dem Vergleich beide Operanden nach `int` umgewandelt
- weil dies ohne Wertverlust geschehen kann
- ↪ hier werden zwei `signed`-Werte verglichen
- ↪ ein `unsigned int` würde nicht umgewandelt werden!



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`



1 C-Quiz Teil I

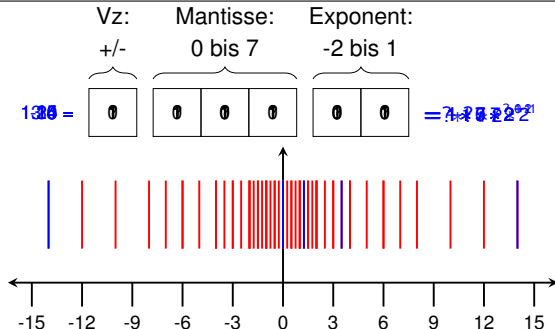
2 Festkommaarithmetik

3 Softwareentwurf

4 Hinweise zur Aufgabe: Filter



Fließkommazahlen



IEEE 754

- Noch komplexer:
 - normalisierte/denormalisierte Darstellung
 - Rundung, Fehlersemantik, ...
 - NaN, ∞ , ...
- <https://ieeexplore.ieee.org/document/4610935/>

```
float func(void){  
    volatile float a = 23.42;  
    volatile float b = 12.34;  
    return a * b;  
}
```

```
func:  
    push    {r7, lr}  
    sub     sp, #8  
    add     r7, sp, #0  
    ldr     r3, [pc, #28] ; float a  
    str     r3, [r7, #4]  
    ldr     r3, [pc, #28] ; float b  
    str     r3, [r7, #0]  
    ldr     r2, [r7, #4]  
    ldr     r3, [r7, #0]  
    adds   r0, r2, #0 ; Param 1  
    adds   r1, r3, #0 ; Param 2  
    bl     3a6c <__aeabi_fmul>  
    adds   r3, r0, #0  
    adds   r0, r3, #0  
    mov    sp, r7  
    add    sp, #8  
    pop    {r7, pc}
```

- Setup
 - Plattform: ARM Cortex-M0+
 - Compiler: arm-gcc
- Funktion `__aeabi_fmul`: 300 Zeilen Assembler
- Keine Fließkommaeinheit (engl. floating-point unit, FPU) vorhanden
- Emulation der **Fließkommaarithmetik in Software**



- Kaum ein Mikrocontroller hat eine *Fließkommaeinheit*
- *Kein EAN* für Fließkommazahlen
 - ↪ *Festkommaarithmetik* mit Ganzzahlen
- Zahlenformat häufig in Q-Notation [2] angegeben
- $Qm.n$ ↪ Festkommazahl mit
 - m Bit vor dem Komma, n nach dem Komma, ein Vorzeichenbit
 - Wertebereich: $[-2^m, 2^m - 2^{-n}]$
 - Auflösung: 2^{-n}
- Implementierung für Übungsaufgabe *vorgegeben*

Implementierung als Integer

↪ passendes Q-Format ist **anwendungsspezifisch**



von Fließkomma nach Qm.n

1. Multiplikation mit 2^n
2. Runden auf die nächste Ganzzahl

von Qm.n nach Fließkomma

1. Umwandlung in Fließkommazahl \rightsquigarrow cast
2. Multiplikation mit 2^{-n}



■ Addition und Subtraktion wie bei Ganzzahlen

Addition

```
1 int32_t    a = ...;  
2 int32_t    b = ...;  
3 int32_t result = a + b;  
4
```

Subtraktion

```
1 int32_t    a = ...;  
2 int32_t    b = ...;  
3 int32_t result = a - b;  
4
```



- Braucht Zwischenergebnis von doppelter Bitbreite

Multiplikation

```
1 #define K (1 << (n - 1))
2 int32_t a = ...;
3 int32_t b = ...;
4 int64_t temp = (int64_t) a * (int64_t) b;
5     temp += K;
6 int32_t result = temp >> n;
7
```

Division

```
1 int32_t a = ...;
2 int32_t b = ...;
3 int64_t temp = (int64_t) a << n;
4     temp += b / 2;
5 int32_t result = temp / b;
6
```

- Siehe Implementierung in `fixedpoint.c`
- **Vorsicht: Rundungsfehler durch Transformationen**



1 C-Quiz Teil I

2 Festkommaarithmetik

3 Softwareentwurf

4 Hinweise zur Aufgabe: Filter



Modifizierbarkeit: lokale Veränderbarkeit

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

Effizienz: optimaler Betriebsmittelbedarf

- wird häufig zu früh berücksichtigt

Verlässlichkeit: lange Zeit funktionsfähig ohne menschlichen Eingriff

- gutmütiges Ausfallverhalten
- muss von Anfang an eingeplant sein!

Verständlichkeit: Isolierung von

- Daten
- Algorithmen



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]

1. Identifiziere **Objekte** und deren Attribute
2. Identifiziere **Operationen** jedes Objekts
3. Lege **Sichtbarkeit** fest
4. Lege **Objektschnittstellen** fest
5. **Implementiere** Objekte



- $\hat{x}[\kappa]$ Schätzung, α 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 (σ_w^2 Prozessvarianz, T Abtastintervall, σ_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 α
 - 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;  
5
```



2. Operationen identifizieren

- Herangehensweise:
 - **Verbenextraktion**
 - für kleinere Probleme: *Intuition*
- Leben eines Objekts:
 1. Initialisierung \rightsquigarrow Betriebsmittel anfordern
 2. Verwendung
 3. Beseitigung \rightsquigarrow 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. `moduleextern`
- 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);  
10
```



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
20
```



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 }
12
```



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 }
10
```





KISS – Keep it Small and Simple!

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



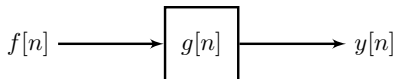
1 C-Quiz Teil I

2 Festkommaarithmetik

3 Softwareentwurf

4 Hinweise zur Aufgabe: Filter





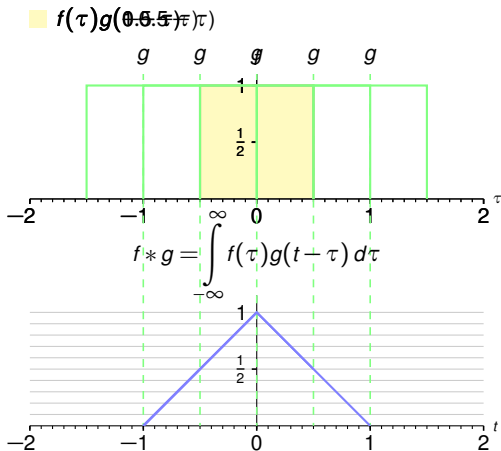
- Objekte identifizieren (z.B. Eingaben)
- Implementierung der Filterung durch **Faltung** (engl. convolution) mit Impulsantwort
 - f Signalwerte, g Filterwerte

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] \cdot g[n-m] \quad (10)$$

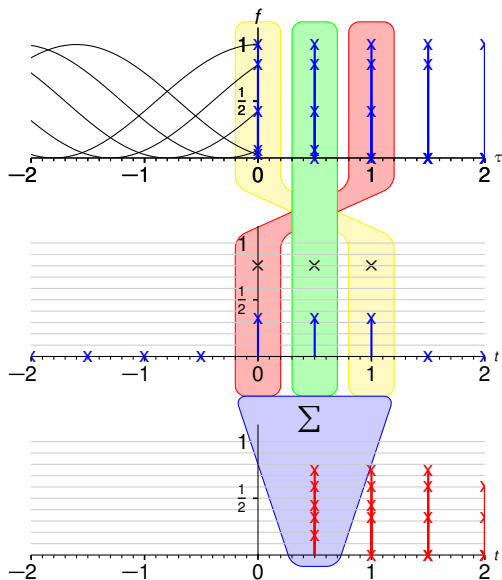
- Impulsantwort: Reaktion des Systems auf Dirac-Impuls
- Oft Transformation in Frequenzbereich \rightsquigarrow verringert Rechenbedarf
- Zunächst Verwendung von `float`, anschließend **Festkommaformat**



Beispiel: Faltung



Beispiel: Diskrete Faltung



$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] \cdot g[n - m]$$

- **Einzelnen Filterschritt** implementieren (kein Burst-Filter)
- Verwendung von **Q-Notation**
- *Wichtig für die späteren FAIL* Fehlerinjektionsexperimente*
 1. Berechnungsdauer möglichst kurz (Fehlerinjektion dauert sehr lange)
 2. FAIL* hat keine Unterstützung für `float`
 3. Keine Verwendung von `printf` bei Fehlerinjektion





Grady Booch.

Software Engineering with Ada.

The Benjamin/Cummings Publishing Company, Inc., 2nd edition, 1987.



Erick L. Oberstar.

Fixed-point representation & fractional math.

Technical report, Oberstar Consulting, August 2007.

