Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Florian Franzmann, Martin Hoffmann

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

11. Juni 2012

Überblick

Ziele des Softwareentwurfs

- 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!

Fragestellung

Wie komme ich von der Beschreibung zur Software?

Objektorientierter/Objektbasierter Entwurf [1]

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

Beispielbeschreibung – α-Filter

- $x[\kappa]$ 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):

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

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

1. Objekte und Attribute identifizieren

- Herangehensweise:
  - Hauptwortextraktion aus Anforderungsdokument
  - für kleinere Probleme: *Intuition*
- Was ist das Objekt? $\leadsto$ Filter
- Attribute? Welche Information brauche ich für jeden Filterschritt?
  - Schätzung aus der Vorrunde $\hat{x}[\kappa - 1]$
  - Filterparameter α
  - aktuellen Messwert $y[\kappa]$ $\leadsto$ 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 sollen Benutzer mit dem Filter machen?
  - Filter initialisieren
  - Filterschritt ausführen
  - Schätzwert erfragen
  - Betriebsmittelfreigabe nicht notwendig

3. Sichtbarkeit festlegen

- in C nur eingeschränkt möglich
  - modulintern vs. modulextern
- Leitfaden: möglichst wenig sichtbar machen
- bei unserem Filter sind öffentlich:
  - Initialisierung
  - Filterschritt
  - Schätzung abfragen
- alle anderen Operationen modulintern

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

$$\hat{x}[0] = 0 \tag{5}$$

$$\lambda = \sigma_w \cdot T^2/\sigma_v \tag{6}$$

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

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; }
```

5. Implementierung – Filterschritt

$$x[\kappa] = x[\kappa - 1] + \alpha \cdot \quad (8)$$

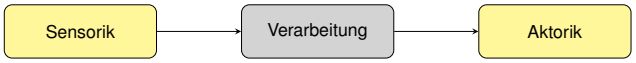
(9)

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  {
9    return filter->x;
10 }
```

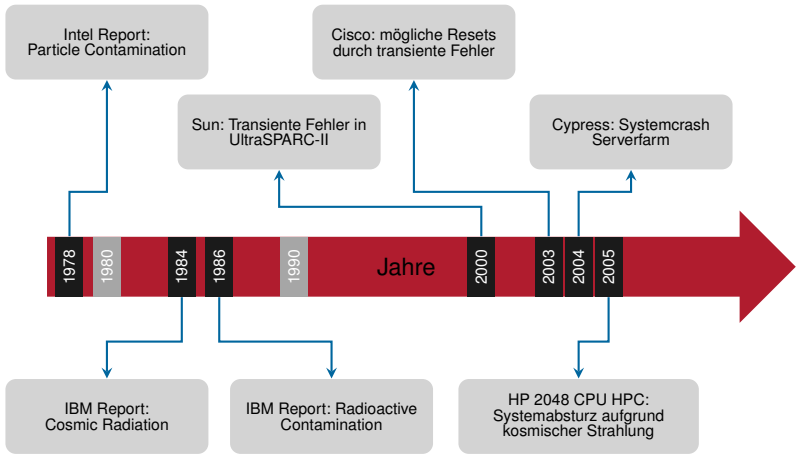

Table of Contents

Fehlertoleranz in eingebetteten Systemen



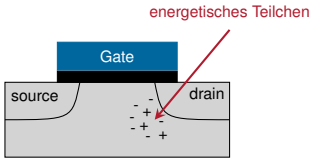
- Hohe Zuverlässigeits- und Sicherheitsanforderungen
  - Safety Integrity Level (SIL), ISO26262, . . .
  - Einsatz von Fehlertoleranztechniken
- aber auch hohe *Kostensensitivität*
  - Trend zu Multiapplikationssystemen
  - Einsatz von Mehrkernarchitekturen

Auswirkungen transienter Fehler



Fehlermodell - Transienter Hardwarefehler

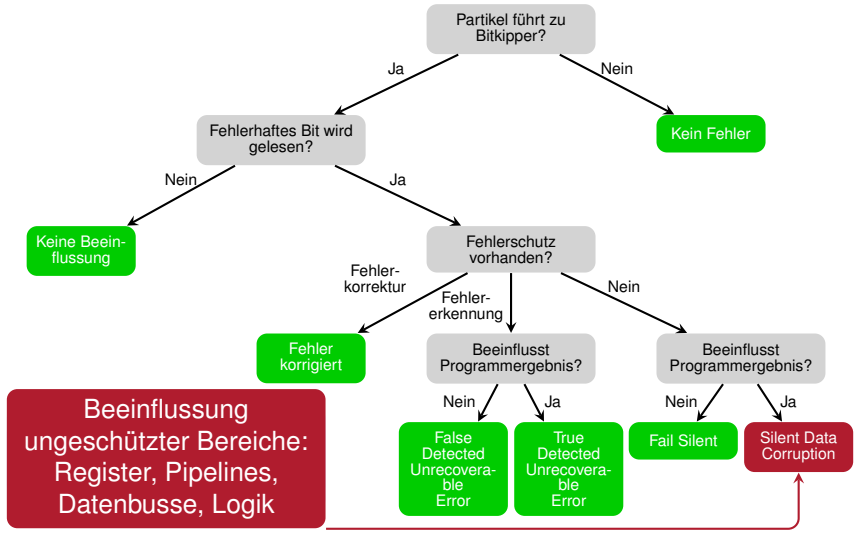
- Steigende Fehleranfälligkeit durch sinkende Strukturgrößen
- Transienter Hardwarefehler (Single Event Upset, Soft-Error)
- Verursacht durch:
  - Ionisierende, elektromagnetische Strahlung
  - Spannungsschwankungen
  - Abgesenkte Versorgungsspannung
  - Rauschen, Übersprechen auf Leitungen



Auswirkung transienter Fehler

- Beeinflussung von:
  - Registerinhalten
  - Berechnungen der ALU
  - Daten oder Instruktionen auf dem CPU Bus
  - Daten auf dem Speicher- oder Peripheriebus

Auswirkungen transienter Fehler



Transiente Fehler

- Aktuelle Hardware kann bestimmte Zuverlässigkeitsanforderungen nicht mehr erfüllen!

International Roadmap for Semiconductors 2002:

"Below 100 *nm*, single event upsets severely impact field-level product reliability, not only for memory, but for logic as well."

Implications of microcontroller software on safety-critical automotive systems (Infineon 2008):

"Probability of failures per hour for usual microcontroller core system is not reaching SIL3 requirements."

- Chiphersteller empfehlen Einsatz von geeigneten Gegenmaßnahmen

N-fach modulare Hardware als Gegenmaßnahme

- Sicherheitskritische Systemkomponenten sind *mehrfach verbaut*:
- Einsatzgebiet: Maskierung von *Hardwarefehlern*
- Hardware NMR $\rightsquigarrow$ Toleranz *permanenter* HW Fehler/Ausfälle

Computers in Spaceflight: The NASA Experience[1]

"(The Space Shuttle's) five general-purpose computers have reliability through *redundancy*, *rather than the expensive quality control* employed in the Apollo program. Four of the computers, each loaded with identical software, operate in what is termed the "redundant set" during critical mission phases such as ascent and descent. The fifth (. . .) is a backup. The four actuators that drive the hydraulics at each of the aerodynamic surfaces are also redundant, as are the pairs of computers that control each of the three main engines."

[1] `http://history.nasa.gov/computers/Ch4-4.html`

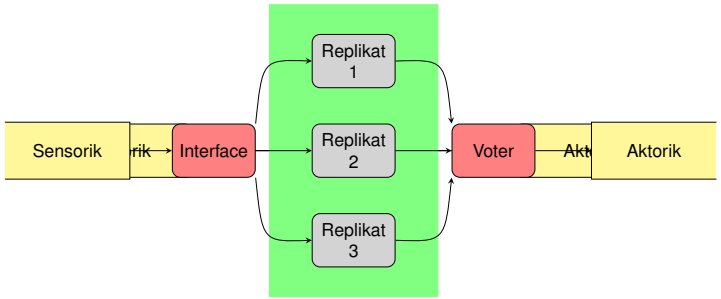
N-fach modulare Hardware (Forts.)

- “Klassische” unkomplizierte Lösung (⇝ *straightforward*)
- Vorteil:
  - Einfache, dennoch effektive Umsetzung
  - Toleranz transienter und *permanenter* Hardwarefehler
- Nachteile:
  - enormer Kostenaufwand im Sinne von:
    - Platz
    - Energie
    - Gewicht
    - Geld...
  - Keine Selektivität (Multiapplikation)
- Dennoch vorgeschrieben für hochsicherheitskritische Systeme:
  - Flugzeuge, Raumfahrtzeuge, Atomkraftwerke, etc.

N-fach modulare Software

- Teure NMR-Hardware in vielen Systemen nicht umsetzbar
  - ⇝ Hohe Kostensensitivät im Automobilbereich
- Lösung: Softwarebasierte NMR
  - Mehrfache Ausführung sicherheitskritischer Softwarekomponenten
  - Vergleich der Ergebnisse (Voting)
  - Idealerweise unter Einhaltung der ursprünglichen Schnittstelle
- Aber: Nur Maskierung *transienter* Hardwarefehler

Klassische “Triple Modular Redundancy” (TMR)



- Schnittstelle sammelt Eingangsdaten (Replikationsdeterminismus)
- Verteilt Daten und aktiviert Replikate
- Mehrheitsentscheider (Voter) wählt Ergebnis
- Ergebnis wird an Aktuator versendet

Redundanzbereich

(In dieser Übung:) Ausschließlich Replikatausführung.

Table of Contents

TMR – Komponenten

Interface

1. Emuliert ursprüngliche Schnittstelle
2. Verteilt und verwaltet Eingabedaten
3. Erzeugt Replikate, führt diese seriell/parallel aus
4. Vergleicht Ergebnisse (2oo3-Entscheidung)

Replikate

1. Erhalten Eingabedaten vom Interface
2. Führen kritischen Code aus
3. Leiten Ergebnisse zurück zum Interface/Voter

Implementierungshinweise

- Gestalten Sie eine möglichst generische TMR-Ausführungsmagie
- ⇝ Replikat-Implementierung als Funktionspointer übergeben
- ⇝ Ein-/Ausgabedaten in generischen Datenstrukturen
- ⇝ Zustand in generischer Datenstruktur

tmr.h

```
1  #include "tmr_config.h" // user defined data types
2
3  typedef void (*)(State_Data_t *, Input_Data_t *,
4    Output_Data_t *) PImpl;
5
6  void tmr_init(State_Data_t *state);
7
8  /** \return -1 if error not recoverable */
9  int tmr_execute(Input_Data_t *input,
10   Output_Data_t *output, PImpl impl);
```

Teilaufgabe Serielle TMR (`tmr_serial.c`)

- Implementiert `tmr.h`-Schnittstelle
- Verwaltet `State_Data_t`-Objekte (global, statisch)
- Initialisiert Replikatzustände per `tmr_init()`
- Führt `impl()` dreifach redundant aus (`tmr_execute()`)
- Vergleicht `impl()`-Ergebnisse, kopiert in `output`

Teilaufgabe

- Überlegen Sie sich, was Zustand, Eingabe, Ausgabe ist
- Testen Sie ihre Implementierung mit ihrem Festkomma-α-β-Filter ⇝ 1000 Filterschritte

Teilaufgabe: PThreads-TMR (`tmr_pthreads.c`)

Teilaufgabe

Implementieren Sie die Replikatausführung nun parallel mit Hilfe von PThreads!

- Interface `tmr_init()`
  1. Erstellt und startet initial Replikat-Threads
  2. Sonst wie serieller Fall
- Interface `tmr_execute()`
  1. Verteilt und verwaltet Eingabedaten
  2. Signalisiert Replikate
  3. Wartet auf Ergebnissignal
  4. Vergleicht Ergebnisse (2oo3-Entscheidung)

Replikat-Thread

1. Führt `impl()` aus
2. Wartet auf Startsignal
3. Signalisiert Abschluss der Codeausführung an `tmr_execute()`
4. Weiter bei 1.

Teilaufgabe: PThreads-TMR (Forts.)

- TMR-Ausführung mit Hilfe von PThreads
- PThreads: Posix Standard für Threads
- Weit verbreitet: Linux, OSX, QNX, Solaris, eCos, u.v.m
- Bietet (C-) Funktionen zur Verwaltung und Steuerung von Threads
- Besteht aus Schnittstellenbeschreibung und Bibliothek:
- `#include <pthread.h>` ⇝ `libpthread.a`
- Linker muss Bibliothek mit Einbinden:
  - GCC: `-lpthread`
  - CMake: `target_link_libraries(<targetname> pthread )`

Teilaufgabe: PThreads-TMR (Forts.)

- Erzeugen eines PThread

```
1 int pthread_create(pthread_t *thread_id,
2   const pthread_attr_t *attributes,
3   void *(thread_func)(void *),
4   void * arguments);
```

- Terminierung eines PThread
  - nach der Rückkehr aus `thread_func`
  - durch expliziten Aufruf von `pthread_exit(void* status)`
- Warten auf Terminierung im Ursprungsprogramm
  - `int pthread_join(pthread_t thread, void **status_ptr)`
- Identität des Threads
  - `int pthread_self()`
  - Vergleich: `int pthread_equal(pthread_t t1, pthread_t t2)`

Beispiel: PThreads-API

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  void* myfunc(void * arg){
5   char* s = (char*)(arg);
6   printf(''Hi, %s!\n'', s);
7  }
8
9  int main(void){
10  char s[] = ''Dude'';
11  char t[] = ''Tux'';
12  pthread_t t1, t2;
13  int status;
14  pthread_create(&t1, 0, myfunc, (void*)s);
15  pthread_create(&t2, 0, myfunc, (void*)t);
16  pthread_join(t1, (void**) &status);
17  pthread_join(t2, (void**) &status);
18  puts(''done.'');
19
20  return 0;
21 }
```

Beispiel: PThreads-API (Forts.)

```
faui41l ~ % gcc pttest.c -lpthread
faui41l ~ % for i in {1..5}; do ./a.out; done
Hi, Dude!
Hi, Tux!
done.
Hi, Tux!
Hi, Dude!
done.
Hi, Tux!
Hi, Dude!
done.
Hi, Dude!
Hi, Tux!
done.
Hi, Dude!
Hi, Tux!
done.
```

Signale mit PThreads

- Zur Signalisierung dienen Condition-Variablen

Initialisierung

```
pthread_mutex_t mutex; pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
pthread_mutex_init(&mutex, NULL);
```

Thread 1

```
pthread_cond_wait(&cond, &mutex);
```

Thread 2

```
pthread_cond_broadcast(&cond);
```

Aufräumen

```
pthread_mutex_destroy(&mutex)
pthread_cond_destroy(&cond)
```

Aufgabenstellung PThreads-TMR (Forts.)

Ausführliche Dokumentation

```
man 3 pthread_{cond_wait,cond_init,cond_signal,\\
mutex_init,mutex_unlock,create,cancel}
```

Aufgabenstellung PThreads-TMR (Forts.)

Teilaufgabe c

Erweitern Sie Ihre Interface Implementierung so, dass Replikate nach einer konfigurierbaren maximalen Ausführungszeit *abgebrochen* werden!

Ausführliche Dokumentation

```
man 3 pthread_cond_timedwait
```

Literatur

[1] Grady Booch.
Software Engineering with Ada.
The Benjamin/Cummings Publishing Company, Inc., 2nd edition, 1987.

Fragen?