

Gliederung

1 Überblick

2 Fallbeispiel Quadrokopter

- Kontrolliertes Objekt
- Kontrollierendes Rechensystem
- Zusammenspiel
- Rolle der Echtzeitanwendung

3 Programmunterbrechung

- synchron vs. asynchron
- Ausnahmebehandlung
- Zustandssicherung

4 Zusammenfassung

Fragestellungen

- Welche Berührungspunkte gibt es zwischen dem physikalischen Objekt und dem kontrollierenden Echtzeitsystem?
 - Woher stammen die **Terminvorgaben** für das Echtzeitrechensystem?
- Welche Rolle spielt überhaupt die **Echtzeitanwendung**?
 - Wie ist sie in das Echtzeitrechensystem eingepasst?
 - Welche Elemente gehören noch zum kontrollierenden Rechensystem?
- Was beeinflusst das Laufzeitverhalten der Echtzeitanwendung?
 - Was muss man für die Beurteilung der Rechtzeitigkeit betrachten?
 - Welche Rolle spielen beispielsweise **Unterbrechungen**?
 - Wie hoch sind die **Verwaltungsgemeinkosten** von Unterbrechungen?

Gliederung

1 Überblick

2 Fallbeispiel Quadrokopter

- Kontrolliertes Objekt
- Kontrollierendes Rechensystem
- Zusammenspiel
- Rolle der Echtzeitanwendung

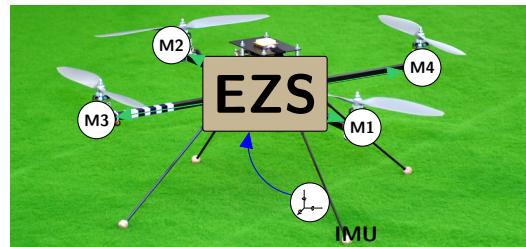
3 Programmunterbrechung

- synchron vs. asynchron
- Ausnahmebehandlung
- Zustandssicherung

4 Zusammenfassung

Aufbau des Demonstrators

Eine elementare Kontrollsleife: Die Fluglageregelung



Quadrokopter sind **inhärent instabil** \sim ständige, aktive Kontrolle

Aufgabe des Echtzeitrechensystems: Fluglageregelung (Stabilisierung)

- Bewegung im Raum bestimmen (engl. **inertial measurement unit**)
- Vorgabe der Motor- und damit der Rotordrehzahl

\sim Setzt sich zusammen aus: **Objekt, Rechensystem, Anwendung**

Kontrolliertes Objekt

Schrittfunktion (engl. *step function*) und Antwortfunktion (engl. *response function*)

Die **Schrittfunktion** verändert den Zustand des kontrollierten Objekts, die Zustandsänderung wird durch die **Antwortfunktion** beschrieben.

Eine **Änderung** der **Rotordrehzahl** des Quadrokopter beeinflusst dessen **Lage im Raum**, bis ein **Gleichgewicht** zwischen Sollzustand und tatsächlichem Zustand hergestellt wurde:

- Wie lange dauert es bis zum Gleichgewicht? \sim **Objektdynamik**
 - Gewicht, Leistungsfähigkeit der Motoren, Bauart der Rotorblätter, ...

Zeitparameter zur Charakterisierung der Schritt-/Antwortfunktion:

d^{object} Zeitdauer bis sich die Lage des Quadrokopter zu ändern beginnt

- hervorgerufen durch die (initiale) Trägheit des Objektes
- auch als Prozessverzögerung (engl. *process lag*) bezeichnet

d^{rise} Zeitdauer bis zum (erneuten) Gleichgewicht

Kontrollierendes Rechensystem

Echtzeitrechensystem

Die Lage des Quadrokopter wird zyklisch abgetastet, um Abweichungen der aktuellen Lage von der Gleichgewichtslage zu erkennen:

d^{sample} Zeitabstand (konstant) zwischen zwei Abtastungen

- analoge auf digitale Werte abbilden \sim A/D-Wandlung
- Faustregel: $d^{sample} < (d^{rise}/10)$
 - quasi-kontinuierliches Verhalten des diskreten Systems

f^{sample} Abtastfrequenz, entspricht $1/d^{sample}$

Abweichung (Ist-/Sollwert) bestimmen und dem Regelungsalgorithmus zur Berechnung des neuen Stellwertes zuführen:

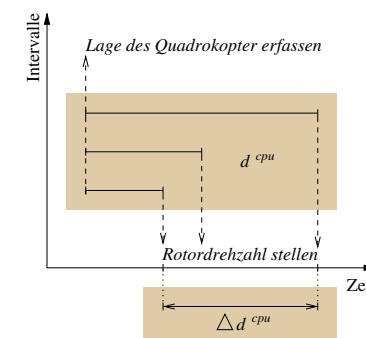
d^{cpu} Zeitdauer bis zur Ausgabe des neuen Stellwertes

- digitale auf analoge Werte abbilden \sim D/A-Wandlung
- Randbedingung: $d^{cpu} < d^{sample}$

Δd^{cpu} Differenz zwischen Minimum und Maximum von d^{cpu}

Kontrollierendes Rechensystem (Forts.)

Schwankung (engl. *jitter*) in den Messergebnissen



d^{cpu} ist trotz konstantem Rechenaufwand zur Stellwertbestimmung variabel

- verdrängende Einplanung
- überlappende Ein-/Ausgabe
- Programmunterbrechungen
- Busüberlastung, DMA

Δd^{cpu} fügt Unschärfe zum Zeitpunkt der Lagebestimmung hinzu

- bewirkt zusätzlichen Fehler
- beeinträchtigt die Dienstgüte

• unbekannte variable Verzögerungen können bei der Regelung nicht kompensiert werden, aber bekannte konstante Verzögerungen

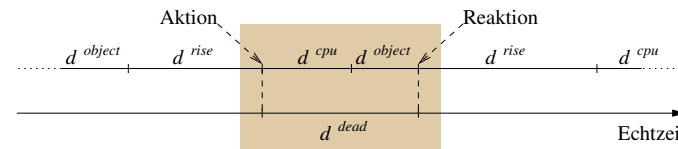
- Randbedingung: $\Delta d^{cpu} \ll d^{cpu}$

Kontrolliertes Objekt \leftrightarrow Kontrollierendes Rechensystem

Totzeit des offenen Regelkreises

d^{dead} Zeitintervall zwischen Start der Aktion zur Stellwertberechnung und Wahrnehmung einer Reaktion nach erfolgter Steuerung

- setzt sich zusammen aus d^{cpu} und d^{object} , d.h.:
 - ① der Implementierung des kontrollierenden Rechensystems
 - ② der Dynamik des kontrollierten Objektes



- beeinträchtigt Güte und **Stabilität** der Kontrollsleife
 - insbesondere in Anbetracht der mit d^{cpu} gegebenen Varianz
- gibt einem relative Ungewissheit über die erzielte Wirkung

Zeitbedarf im kontrollierenden Rechensystem

Welche Komponenten benötigen wie viel Zeit?

Häufig ist eine eigenständige Beurteilung des Zeitbedarfs nicht möglich, Herstellerangaben ermöglichen die Abschätzung des **schlimmsten Falls**.

d^{imu} Gyroskop ITG-3200 – Abtastrate: 4 Hz – 8 kHz [1]

d^{adc} Infineon TriCore ADC: 280 ns – 2,5 μ s @ 10 Bit [2]

d^{irq} Infineon TriCore Arbitrierung: 5 - 11 Takte @ 150 MHz [2]

d^{OS} CiAO OS Fadenwechsel: \leq 219 Takte @ TriCore (50 MHz) [3]

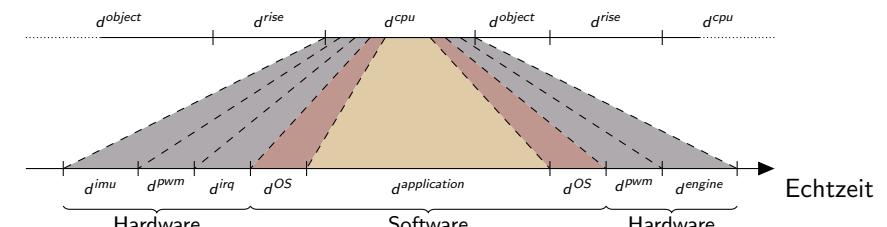
Alleine die **Anwendung** kann (fast) komplett kontrolliert werden.¹

¹ Lässt man zugelieferte Bibliotheksfunktionen oder zugekauftes Codegeneratoren außer Acht.

Vorgänge im Rechensystem – Echtzeitanwendung

Aus welchen Komponenten setzt sich d^{cpu} zusammen?

Das kontrollierende Rechensystem setzt sich aus verschiedenen Sensoren, Peripherie-Elementen und Softwarekomponenten zusammen.



☞ Alle Komponenten eines Echtzeitystems müssen bedacht werden!

Sensoren/Aktoren Abtastrate ($\sim d^{imu}$), Motorleistung ($\sim d^{engine}$)

Mikrocontroller Signalverarbeitung ($\sim d^{pwm}$), IRQ ($\sim d^{irq}$)

Betriebssystem Unterbrechungslatenz, Kontextwechsel ($\sim d^{OS}$)

Anwendung Steuerung, Regelung ($\sim d^{application}$)

Gliederung

1 Überblick

2 Fallbeispiel Quadrokopter

- Kontrolliertes Objekt
- Kontrollierendes Rechensystem
- Zusammenspiel
- Rolle der Echtzeitanwendung

3 Programmunterbrechung

- synchron vs. asynchron
- Ausnahmebehandlung
- Zustandssicherung

4 Zusammenfassung

Zeitliches Verhalten von Echtzeitanwendungen

Welche Elemente beeinflussen das zeitliche Verhalten einer Echtzeitanwendung?

- Welche Elemente müssen betrachtet werden?

- Kann man sich auf die Echtzeitanwendung selbst beschränken?
- Kann das Betriebssystem oder das Laufzeitsystem ignoriert werden?
- Wie stark hängt dies vom verwendeten Prozessor ab?

- Auf welcher Ebene muss man diese Betrachtung durchführen?

- Genügt es eine hohe Abstraktionsebene heranzuziehen?
- Wo entscheidet sich das zeitliche Ablaufverhalten?

☞ exemplarische Illustration anhand von Programmunterbrechungen

Synchrone Programmunterbrechung

- unbekannter Befehl, falsche Adressierungsart oder Rechenoperation
- Systemaufruf, Adressraumverletzung, unbekanntes Gerät

Trap ↪ synchron, vorhersagbar, reproduzierbar

- geschieht abhängig vom Arbeitszustand des laufenden Programms:
 - unverändertes Programm, mit den selben Eingabedaten versorgt
 - auf ein und dem selben Prozessor zur Ausführung gebracht
- die Unterbrechungsstelle im Programm ist vorhersehbar

☞ die Programmunterbrechung/-verzögerung ist deterministisch

Unterbrechungsarten

Zwei Arten von Programmunterbrechungen werden unterschieden:

synchron die „Falle“ (engl. *trap*)

asynchron die „Unterbrechung“ (engl. *interrupt*)

Unterschiede ergeben sich hinsichtlich...

- Quelle
- Synchronität
- Vorhersagbarkeit
- Reproduzierbarkeit

☞ Behandlung ist zwingend und grundsätzlich prozessorabhängig

Wiederholung/Vertiefung empfohlen...

- zu Unterbrechungen siehe auch „Betriebssystemtechnik“ [4]

Asynchrone Programmunterbrechung

- Signalisierung „externer“ Ereignisse
- Beendigung einer DMA- bzw. E/A-Operation

Interrupt ↪ asynchron, unvorhersagbar, nicht reproduzierbar

- tritt unabhängig vom Arbeitszustand des laufenden Programms auf:
 - hervorgerufen durch einen „externen Prozess“ (z.B. ein Gerät)
 - ein Ereignis signalisierend
- die Unterbrechungsstelle im Programm ist nicht vorhersehbar

☞ die Programmunterbrechung/-verzögerung ist nicht deterministisch

Trap/Interrupt ~ Ausnahmesituationen

Ereignisse, oftmals unerwünscht aber nicht immer eintretend:

- Signale von der Peripherie (z.B. E/A, Zeitgeber oder „Wachhund“)
- Wechsel der Schutzdomäne (z.B. Systemaufruf)
- Programmierfehler (z.B. ungültige Adresse)
- unerfüllbare Speicheranforderung (z.B. bei Rekursion)
- Einlagerung auf Anforderung (z.B. beim Seitenfehler)
- Warnsignale von der Hardware (z.B. Energiemangel)

Ereignisbehandlung, die problemspezifisch zu gewährleisten ist:

- als Ausnahme während der „normalen“ Programmausführung

Zustandssicherung

Prozessorstatus invariant halten

Hardware (CPU) sichert einen Zustand minimaler Größe²

- Statusregister (SR)
- Befehlszeiger (engl. *program counter*, PC)

Software (Betriebssystem/Kompilierer) sichert den restlichen Zustand

- alle $\left\{ \begin{array}{l} \text{dann noch ungesicherten} \\ \text{flüchtigen} \\ \text{im weiteren Verlauf verwendeten} \end{array} \right\}$ CPU-Register

☞ je nach CPU werden dabei wenige bis sehr viele Daten(bytes) bewegt

²Möglicherweise aber auch den kompletten Registersatz.

Ausnahmebehandlung

Abrupter Zustandswechsel

Programmunterbrechungen implizieren **nicht-lokale Sprünge**:

- vom $\left\{ \begin{array}{l} \text{unterbrochenen} \\ \text{behandelnden} \end{array} \right\}$ zum $\left\{ \begin{array}{l} \text{behandelnden} \\ \text{unterbrochenen} \end{array} \right\}$ Programm

Sprünge (und Rückkehr davon), die **Kontextwechsel** nach sich ziehen:

- erfordert Maßnahmen zur Zustandssicherung/-wiederherstellung
- Mechanismen dazu liefert das behandelnde Programm selbst
 - bzw. eine tiefer liegende Systemebene (Betriebssystem, CPU)

☞ der **Prozessorstatus** unterbrochener Programme muss invariant sein

Prozessorstatus sichern und wiederherstellen

Unabhängigkeit von der Sprachebene der Behandlungsprozedur

... alle dann noch ungesicherten CPU-Register:

Zeile

1:
2:
3:
4:
5:

x86

train:
pushal
call handler
popal
iret

m68k

train:
moveml d0-d7/a0-a6,a7@-
jsr handler
moveml a7@+,d0-d7/a0-a6
 rte

train (trap/interrupt):

- Arbeitsregisterinhalte im RAM sichern (2) und wiederherstellen (4)
- Unterbrechungsbehandlung durchführen (3)
- Ausführung des unterbrochenen Programms wieder aufnehmen (5)

Prozessorstatus sichern und wiederherstellen (Forts.)

Abhängigkeit von der Sprachebene der Behandlungsprozedur

... alle **flüchtigen Register**³ (engl. *volatile register*) der CPU:

x86

```
train:
    pushl %edx
    pushl %ecx
    pushl %eax
    call  handler
    popl  %eax
    popl  %ecx
    popl  %edx
    iret
```

m68k

```
train:
    moveml d0-d1/a0-a1,a7@-
    jsr handler
    moveml a7@+,d0-d1/a0-a1
    rte
```

³ Register, deren Inhalte nach Rückkehr von einem Prozeduraufruf verändert worden sein dürfen: festgelegt in den **Prozedurkonventionen** des Kompilierers.

Prozessorstatus sichern und wiederherstellen (Forts.)

Abhängigkeit von den Eigenschaften des Kompilierers

... alle im weiteren Verlauf verwendeten CPU-Register:

gcc

```
void __attribute__((interrupt)) train () {
    handler();
}
```

__attribute__((interrupt))

- Generierung der speziellen Maschinenbefehle durch den **Kompilierer**
 - zur Sicherung/Wiederherstellung der Arbeitsregisterinhalte
 - zur Wiederaufnahme der Programmausführung
- nicht jeder „Prozessor“ (für C/C++) implementiert dieses Attribut

Aktivierungsblock (engl. *activation record*)

Sicherung/Wiederherstellung nicht-flüchtiger Register (engl. *non-volatile register*)

Türme von Hanoi

```
void hanoi (int n, char from, char to, char via) {
    if (n > 0) {
        hanoi(n - 1, from, via, to);
        printf("schlephe Scheibe %u von %c nach %c\n", n, from, to);
        hanoi(n - 1, via, to, from);
    }
}
```

Aufwand je nach CPU, Prozedur, Kompilierer: gcc -O6 -S hanoi.c

hanoi()-Eintritt

```
pushl %ebp
movl %esp,%ebp
pushl %edi
pushl %esi
pushl %ebx
subl $12,%esp
```

hanoi()-Austritt

```
leal -12(%ebp),%esp
popl %ebx
popl %esi
popl %edi
popl %ebp
ret
```

Für eine Prozedur aufrufende Ebene **inhaltssinvariante Register** der CPU, deren Inhalte jedoch innerhalb einer aufgerufenen Prozedur verändert werden:

gcc/x86 ~ ebp, edi, esi, ebx

Verwaltungsgemeinkosten des schlimmsten Falls

(engl. *worst-case administrative overhead*, WCAO)

Latenz ... bis zum Start der Unterbrechungsbehandlung:

- 1 Annahme der Unterbrechung durch die Hardware
- 2 Sicherung der Inhalte der (flüchtigen) CPU-Register
- 3 Aufbau des Aktivierungsblocks der Behandlungsprozedur

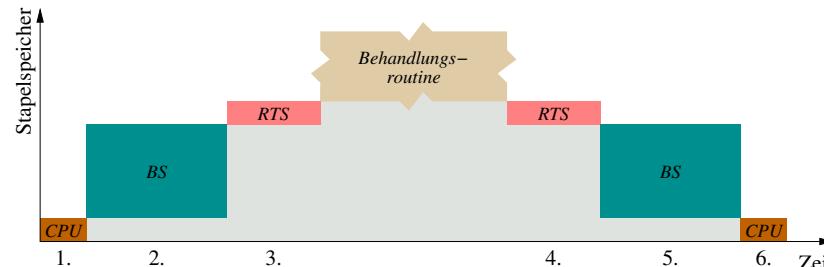
... bis zur Fortführung des unterbrochenen Programms:

- 4 Abbau des Aktivierungsblocks der Behandlungsprozedur
- 5 Wiederherstellung der Inhalte der (flüchtigen) CPU-Register
- 6 Beendigung der Unterbrechung

☞ **Zeitpunkte** und **Häufigkeit** der Gemeinkosten sind i. A. unbestimmt

Verwaltungsgemeinkosten des schlimmsten Falls (Forts.)

Speicherplatz vs. Laufzeit



Systemkonstanten bzw. **Werte mit fester oberer Schranke** sind gefordert:

- CPU resp. Hardware
- Betriebssystem (BS), Laufzeitsystem (engl. *run-time system*, RTS)
- „Anwendung“ (Behandlungsroutine)

Gliederung

1 Überblick

2 Fallbeispiel Quadrokopter

- Kontrolliertes Objekt
- Kontrollierendes Rechensystem
- Zusammenspiel
- Rolle der Echtzeitanwendung

3 Programmunterbrechung

- synchron vs. asynchron
- Ausnahmebehandlung
- Zustandssicherung

4 Zusammenfassung

Resümee

Zusammenspiel Kontrolliertes Objekt \leftrightarrow Kontrollierendes Rechensystem

- die **Objektdynamik** definiert den zeitlichen Rahmen durch Termine
- die Echtzeitanwendung muss diese Termine einhalten
- ihr Anteil am kontrollierenden Rechensystem ist eher gering

Programmunterbrechung in synchroner oder asynchroner Ausprägung

- beeinflussen den Ablauf der Echtzeitanwendung
- Zustandssicherung, Verwaltungsgemeinkosten des schlimmsten Falls

Literaturverzeichnis

[1] INC., I. :

ITG-3200 Product Specification Revision 1.4.

<http://invensense.com/mems/gyro/documents/PS-ITG-3200A.pdf>, 2010. –
Data Sheet

[2] INFINEON TECHNOLOGIES AG (Hrsg.):

TC1796 User's Manual (V2.0).

St.-Martin-Str. 53, 81669 München, Germany: Infineon Technologies AG, Jul. 2007

[3] LOHMANN, D. ; HOFER, W. ; SCHRÖDER-PREIKSCHAT, W. ; STREICHER, J. ; SPINCKYK, O. :

CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems.
In: *Proceedings of the 2009 USENIX Annual Technical Conference*.
Berkeley, CA, USA : USENIX Association, Jun. 2009. –

ISBN 978-1-931971-68-3, S. 215–228

[4] SCHRÖDER-PREIKSCHAT, W. :

Betriebssystemtechnik.

<http://www4.informatik.uni-erlangen.de/Lehre/BST>, 2009. –
Lecture Notes