Echtzeitsysteme

Übungen zur Vorlesung

Nicht-periodische Aufgaben: Extended Scope

Simon Schuster **Peter Wägemann**

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

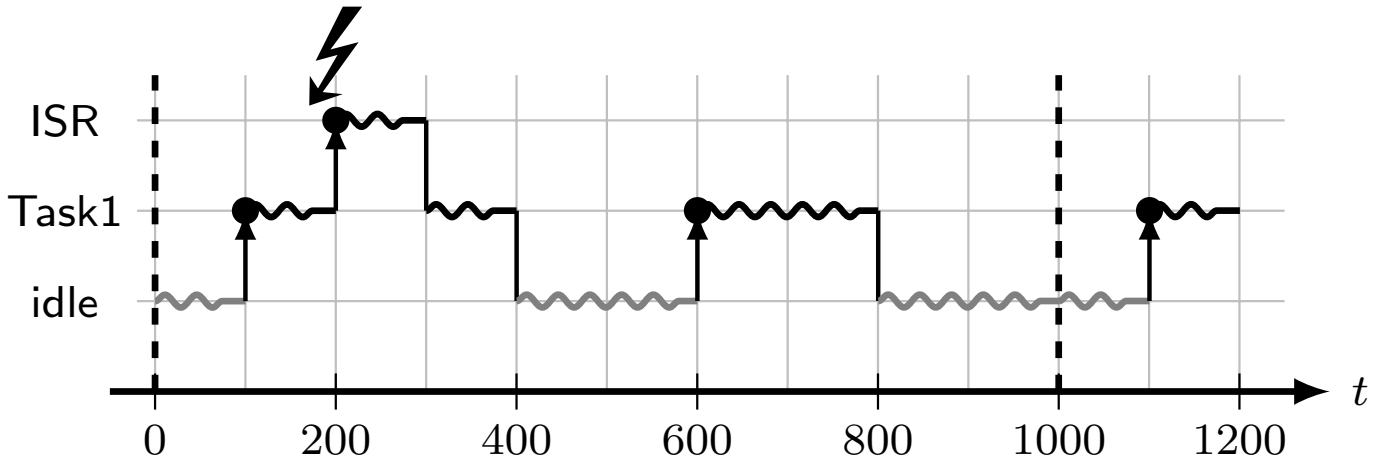
Sommersemester 2022

Übersicht

Begriffe

- *Interrupt*: Hardwareunterstützung für Kontrolltransfer an Interrupt-Handler
- *Interrupt-Handler*: Code der beim Auftreten des Interrupts ausgeführt wird
- *Interrupt-Vektor*: Nummer & Speicheradresse des Interrupt-Handlers
- *Interrupt-Controller*: Hardwareeinheit für Interruptbehandlung
- *Pending Interrupt*: noch nicht abgearbeiteter Interrupt
- *Interrupt-Latenz*: Zeit bis Interrupt erkannt/behandelt wird
- *Geschachtelter Interrupt*

Probleme von Interrupts [1, 2]



- *Prioritätsverletzungen*
- Prellen
  - Entprellung in Soft- oder Hardware
  - Tiefpassfilterung
- Auftrittshäufigkeit
  - Maximale Auftrittsfrequenz (= minimale Zwischenankunftszeit)
  - Soft- oder hardwareseitige Überwachung
- Auftrittszeitpunkte
  - Zeitliche Garantien jederzeit gewährleisten
  - Zugriff auf Ressourcen

Checkliste für Interrupts in (V)EZS [1]

1. *Scheduling*: (WCET-)Analyse muss Interrupts (als Overheads) beachten
2. *Zeitanalyse*
   - Bestimmung der **maximalen Auftrittsfrequenz**
   - WCET-Analyse der Interrupt-Behandlung (in Isolation)
3. *Aufrufgraphen*
   - **Identifikation der Kontexte** in denen Interrupts auftreten könnten
   - ISR → DSR → `cyg_thread_resume()`
4. *Korrektheit des Stacks*
   - Gemeinsamer Stack?
   - Effekte von Interrupts auf Stack-Anordnungen
   - Bestimmung von **Stack-Budgets** (worst-case stack usage)
5. *Korrektheit der Nebenläufigkeiten*
   - Identifikation von Datenstrukturen auf die nebenläufig zugegriffen wird
   - Vermeidung von **Race-Conditions**, Verwendung atomarer Operationen

Interrupt Scheduler [2]

- Detektion und **Behandlung falscher Interrupts** (engl. spurious interrupts)
- Externe Geräte können fehlerhaft sein $\rightsquigarrow$ *Babbling Idiot*
- Software-Lösung
  - Zählen von Interrupts über Zeitintervall
  - Verwendet in Linux [1]
  - Nur möglich wenn WCET(ISR) < minimale Zwischenankunftszeit
    - Detektion von spurious Interrupts
    - Deaktivierung des IRQs
    - Adaptives Pollen von Geräten

      ☞Zusätzlicher Laufzeit-Overhead
- Hardware-Lösung (bevorzugt für harte Echtzeit)
  - **Zählen in Hardware** der Auftrittshäufigkeiten
  - TriCore CPU erlaubt das Zählen von externen Ereignissen (Komparatoren)
  - Überwachung implementierbar
  - Kein zusätzlicher Laufzeit-Overhead (außer Konfigurationsaufwand)

[1] `https://github.com/torvalds/linux/blob/master/kernel/irq/spurious.c`

Übersicht

Rekapitulation der Vorlesung

Nicht-periodische Aufgaben

- Definiert durch $T_i = (i_i, e_i, D_i)$
- *Aperiodische* vs. *sporadische* Aufgabe
- *Mischbetrieb*: periodisch $\longleftrightarrow$ sporadisch/aperiodisch
  - *Dynamische* Einplanung
  - Beeinflussung periodischer Aufgaben?
  - Übernahmeprüfung $\longleftrightarrow$ Antwortzeitminimierung

Nicht-periodische Arbeitsaufträge

- Kaum a-priori Wissen (Zeitpunkt, . . .)
- Herausforderung Mischbetrieb: Erhaltung statischer Garantien
- Abweisung (spor. Aufg.): schwerwiegende Ausnahmesituation

Rekapitulation der Vorlesung (Forts.)

Basistechniken zur Umsetzung

- Unterbrecherbetrieb $\rightsquigarrow$ Bevorzugt nicht-periodische Aufgaben
- Hintergrundbetrieb $\rightsquigarrow$ Stellt nicht-periodische Aufgaben hinten an
- Slack Stealing
  - Idee: Termin ist maßgeblich
    $\rightsquigarrow$ *Verschieben* periodischer Aufgaben möglich
  - *Erfordert Unterbrecherbetrieb*
  - Problem: Schlupfzeit bestimmen
    - Zeitsteuerung (mit Rahmen): einfach $\rightsquigarrow$ $f - x_k$
    - Ereignissteuerung: schwierig $\rightsquigarrow$ dynamische Berechnung
- Zusteller $\rightsquigarrow$ Konvertieren nicht-period. in periodische Aufgaben
  - Spezielle periodische Aufgabe $T_s = (p_s, e_s)$
  - Ausführungsbudget, Auffüllperiode und -regeln
  - Abbildung auf Prioritätswarteschlange (z. B. AJQ)

Rekapitulation der Vorlesung (Forts.)

Periodische Zusteller

- Verschiedene Ausführungen
  z. B.: Polling, Deferrable, Sporadic Server
- Unterscheiden sich im Regelwerk
- i. d. R. für mehrere Aufgaben zuständig

Beispiel: Abfragender Zusteller (Polling Server)

- Periodische Aufgabe $T_P = (p_s, e_s)$
- Budget e_s verfällt
- Im Falle sporadischer Aufgaben schwierig:
  - $p_P \leq \frac{D_s}{2}$, wobei $D_s \leq i_s \rightsquigarrow$ Abtasttheorem

→ hohe Abtastfrequenz, Überlastgefahr

Rekapitulation der Vorlesung (Forts.)

Bandweite-bewahrende Zusteller

- Budget bleibt erhalten
  $\rightsquigarrow$ Verbesserung des Abfragebetriebs
- Regelwerk wird erweitert
  $\rightsquigarrow$ Auffüll- und Konsumregeln
- Betriebssystem (Scheduler) wacht über Budget

Auslegung

- Größe Budget
  $\rightsquigarrow$ Berücksichtigung aller (nicht-)periodischer Aufgaben
- Verbesserung Antwortzeit
  $\rightsquigarrow$ Kombination mit Hintergrundbetrieb

Rekapitulation der Vorlesung (Forts.)

Beispiel: Aufschiebbarer Zusteller (Deferrable Server)

- Verbrauchsregel: verbraucht $\frac{1}{Zeiteinheit}$ Budget bei Tätigkeit
- Auffüllregel: periodisches Auffüllen von e_s mit p_s
- Keine Akkumulation

Achtung: aufschiebbarer Zusteller $\neq$ periodische Aufgabe

- **Double hit**

  $\rightsquigarrow$ Kritischer Zeitpunkt und Auffüllzeitpunkt fallen zusammen
- $\rightsquigarrow$ Störung ist bis zu e_s größer als bei periodischer Aufgabe

Rekapitulation der Vorlesung (Forts.)

Lösungsansatz: Sporadischer Zusteller (Sporadic Server)

- Verschiedene Ausprägungen
- Beansprucht niemals mehr Zeit als periodische Aufgabe

Beispiel: SpSL Sporadic Server (Sprunt, Sha & Lehoczky)

- Verbraucht $\frac{1}{\textit{Zeiteinheit}}$ Budget bei Tätigkeit
- Aufgefüllt wird entsprechend dem Verbrauchsmuster
  - Nächster Auffüllzeitpunkt wird zu Beginn der Tätigkeit bestimmt
  - Aufzufüllendes Budget zum Ende der Tätigkeit
  - $\rightsquigarrow$ Auffüllregeln R1 – R3
- SpSL Sporadic Server

$\rightsquigarrow$ *Menge von Aufgaben* T_i mit $p_i = p_s$ und $\sum e_i = e_s$

Rekapitulation der Vorlesung (Forts.)

Forts.: SpSL Sporadic Server, Auffüllregeln

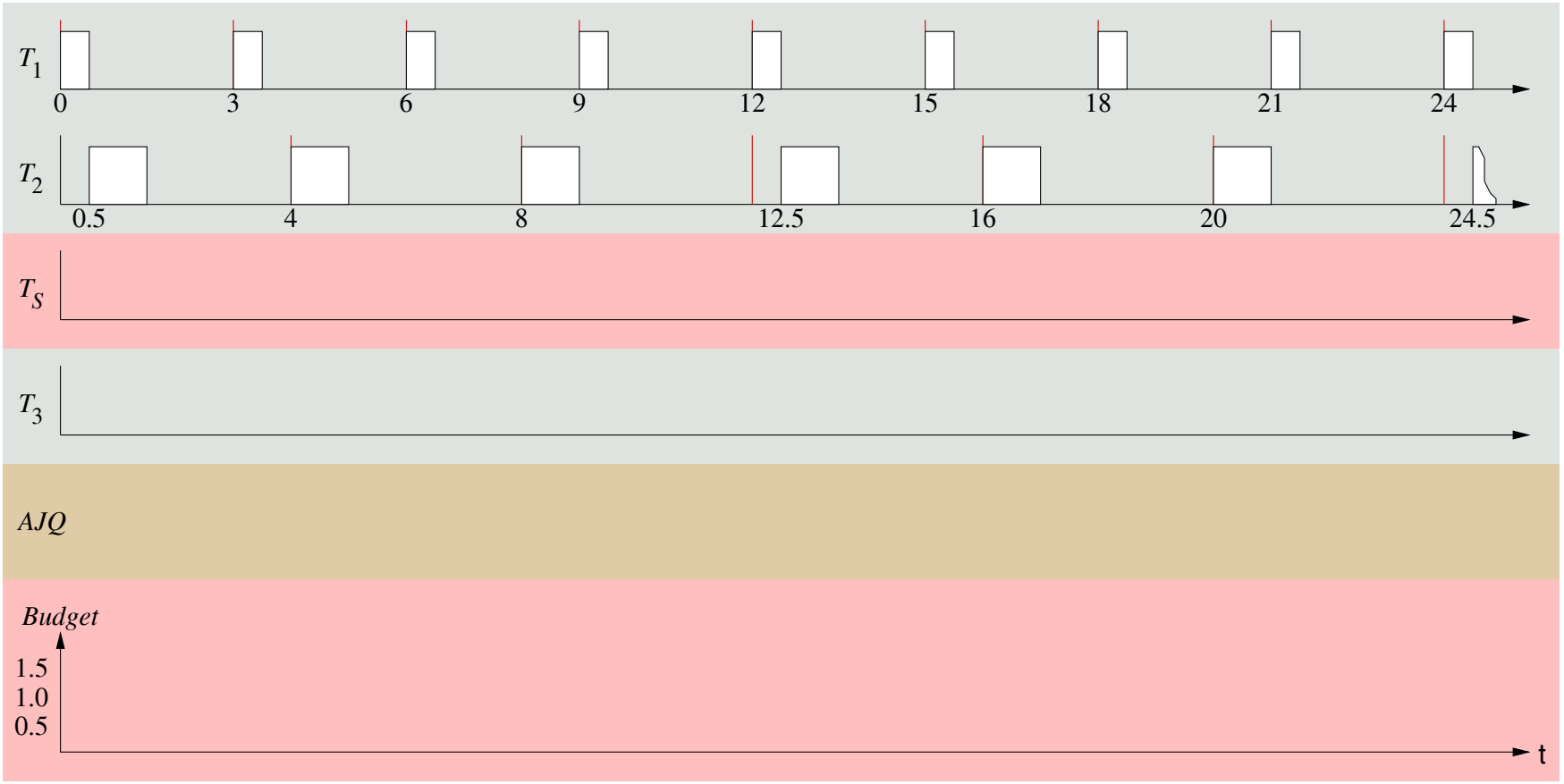
- R1: initiales Budget ist e_s
- R2: Auffüllzeitpunkt $rt_s = t_b + p_s$, wobei:
  - T_s besitzt Budget, dann $t_b = P_s$ wird tätig
  - T_s hat kein Budget, dann $t_b = P_s$ *ist* tätig und T_s *erhält Budget*
- R3: Budgetberechnung
  - Sobald P_s untätig wird oder T_s kein Budget mehr hat
  - Budget für rt_s = Verbrauch von T_s seit t_b

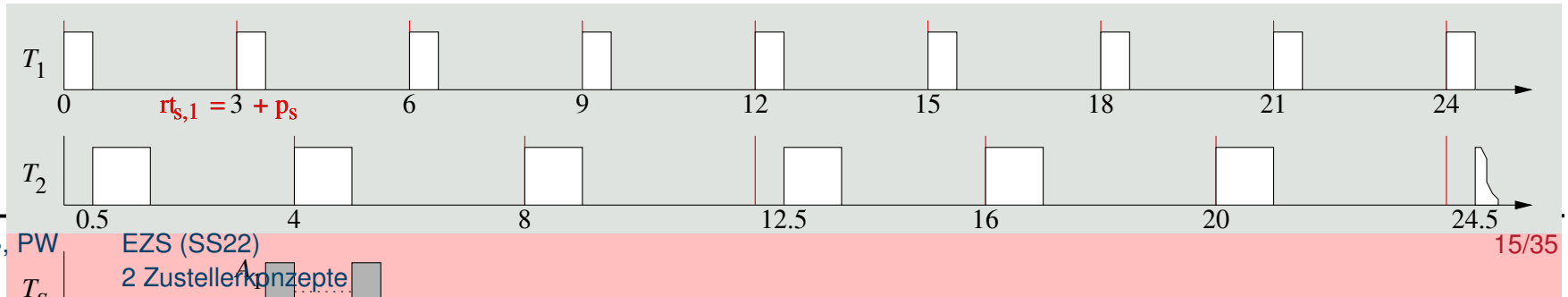
Achtung

- P_s bezeichnet das Tasksystem ab der Priorität s (und höher)
- Im Beispiel: kleinere Zahl $\rightsquigarrow$ höhere Priorität

Beispiel: SpSL

$T_1 = (3, 0.5)$, $T_2 = (4, 1)$, $T_3 = (19, 4.5)$ und $T_s = (5, 1.5)$; RM-Ablaufplanung





Übersicht

Rekapitulation der Vorlesung

Kausalordnung

- Relation: Ursache, Wirkung
- Mehrere Ursache-Wirkungspaare überlappend: Nebenläufigkeit
- Nebenläufigkeit vs. Gleichzeitigkeit

☞ Sequentialisierung von Aufgaben

Rangfolge

- Abhängigkeit von Kontrollfluss $\rightsquigarrow$ Reihenfolge
- Oft in Datenabhängigkeiten begründet
  - Produzent/Konsument Verhältnis
  - Konsumierbare Betriebsmittel (Nachrichten, Interrupts, . . .)
  - Begrenzte Puffer limitieren die Anzahl häufig

☞ Beachtung unterschiedlicher **zeitlicher Domänen**

Rangfolgen (Forts.)

Koordinierung

- Unnötig falls Rangfolge egal
  - Neuester Wert ist ausreichend
- Durch Einplanung $\rightsquigarrow$ analytische Verfahren
  - Periodische Aufgaben $\rightsquigarrow$ Passende Perioden!
  - Ablauftabelle
  - Keine Kontrolle zur Laufzeit
- Durch Kooperation $\rightsquigarrow$ konstruktive Verfahren
  - Periodische und nicht-periodische Aufgaben
  - Synchronisation $\rightsquigarrow$ Vielzahl von Möglichkeiten
  - In zeitgesteuerten Systemen unsinnig!

Übersicht

eCos-Event-Flags[2]

Signalisieren von Ereignissen

- Signale unterstützen *Produzent-Konsument Muster*
- Thread/DSR *signalisiert* Ereignis (z. B. Tastendruck)
  . . . konsumierender Thread *wartet*
- Umsetzung: 32-bit Integer $\rightsquigarrow$ 32 *Einzelsignale* pro Flag
  - Ein Flag erlaubt somit $2^{32} - 1$ Signalkombinationen
  - Threads können auf ein Signalmuster blockierend warten oder pollen

Achtung:

Flags zählen keine Ereignisse! (vgl. HW-Interrupts)

[2] `http://ecos.sourceware.org/docs-latest/ref/kernel-flags.html`

eCos-Event-Flags

- Produzenten/Konsumenten teilen sich eine Flag-Objekt
- Dieses wird von der *Anwendung* bereitgestellt (vgl. Alarmobjekt)
- Flag-Objekt muss initialisiert werden:

```
cyg_flag_init(cyg_flag_t* flag)
```

- Signal(e) im Flag setzen:

```
cyg_flag_setbits(cyg_flag_t* flag, cyg_flag_value_t value)
```

- Bzw. zurücksetzen:

```
cyg_flag_maskbits(cyg_flag_t* flag, cyg_flag_value_t value)
```

- Auf Signal warten/pollen:

```
cyg_flag_value_t cyg_flag_wait/poll(cyg_flag_t* flag,
                                    cyg_flag_value_t pattern,
                                    cyg_flag_mode_t mode);
```

- Warten mit zeitlicher Obergrenze:

```
cyg_flag_value_t cyg_flag_timed_wait(cyg_flag_t* flag,
                                     cyg_flag_value_t pattern,
                                     cyg_flag_mode_t mode,
                                     cyg_tick_count_t abstime);
```

eCos-Event-Flags

- `cyg_flag_value_t pattern` setzt gewünschte Signalkombination
- `cyg_flag_mode_t` legt Weckmuster fest
  - `CYG_FLAG_WAITMODE_AND`: alle konfigurierten Signale müssen aktiv sein; sie bleiben nach Aufwachen gesetzt
  - `CYG_FLAG_WAITMODE_OR`: mindestens eines der konfigurierten Signale muss aktiv sein; alle Signale bleiben nach dem Aufwachen gesetzt
  - `CYG_FLAG_WAITMODE_OR | CYG_FLAG_WAITMODE_CLR`: mindestens eines der konfigurierten Signale muss aktiv sein; alle gesetzten Signale werden nach dem Aufwachen gelöscht

eCos-Event-Flags

Beispiel

```
1  static cyg_flag_t flag0;
2
3  void my_dsr(cyg_vector_t v,
4              cyg_ucount32 c,
5              cyg_addrword_t d){
6    cyg_flag_setbits(&flag0, 0x02);      // 0b00000010
7  }
8
9  void user_thread(cyg_addr_t data){
10   while(true) {
11     cyg_flag_value_t val;
12     val = cyg_flag_wait(&flag0, 0x22, // 0b00100010
13                         CYG_FLAG_WAITMODE_OR
14                         | CYG_FLAG_WAITMODE_CLR);
15     ezs_printf("Event! %" PRIu32 "\n", (cyg_uint32) val);
16     // Prints "Event! 2"
17   }
18 }
19
20 void cyg_user_start(void){
21   ...
22   cyg_flag_init(&flag0);
23   ...
24 }
```

eCos-Mailboxes[3]

- Zwischen Threads können *Nachrichten* versendet werden
- Konsument erzeugt einen Briefkasten (engl. mailbox) fester Größe
- Produzenten legt Nachrichten dort ab
  - Inhalt: Zeiger auf beliebige Datenstruktur
  - Konsument kann auf *Nachrichtenempfang* blockieren
  - Produzent blockiert, falls Briefkasten *voll*
  - Aber auch *nicht-blockierende* Aufrufvarianten

[3] `http://ecos.sourceware.org/docs-latest/ref/kernel-mail-boxes.html`

Versenden von Nachrichten - eCos Mail Boxes[4]

- Mailbox anlegen:
```
void cyg_mbox_create(cyg_handle_t* handle, cyg_mbox* mbox);
```
- Nachricht verschicken:
```
cyg_bool_t cyg_mbox_put(cyg_handle_t mbox, void* item);
```
- Nachricht empfangen:
```
void* cyg_mbox_get(cyg_handle_t mbox);
```

- Empfang *und* Versand können blockieren
- `*try*`-Versionen: Würde ich blockieren?
- `*timed*`-Versionen: Blockieren für bestimmte Zeit

→ Selbststudium!

[4] http://ecos.sourceware.org/docs-latest/ref/kernel-mail-boxes.html

Versenden von Nachrichten – Beispiel

- Initialisierung:

```
1 static cyg_handle_t mailbox_handle;
2 static cyg_mbox     mailbox;
3 void cyg_user_start(void) {
4   cyg_mbox_create(&mailbox_handle, &mailbox);
5   ...
6 }
```

- Produzent (Sender):

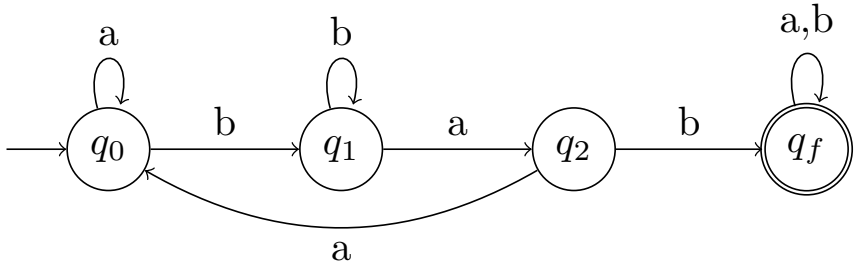
```
1 void producer_entry(cyg_addrword_t data) {
2   ...
3   cyg_mbox_put(mailbox_handle, &my_message);
4   ...
5 }
```

- Konsument (Empfänger):

```
1 void consumer_entry(cyg_addrword_t data) {
2   ...
3   void *message = cyg_mbox_get(mailbox_handle);
4   ...
5 }
```

Übersicht

Aufgabe 6: Extended Scope

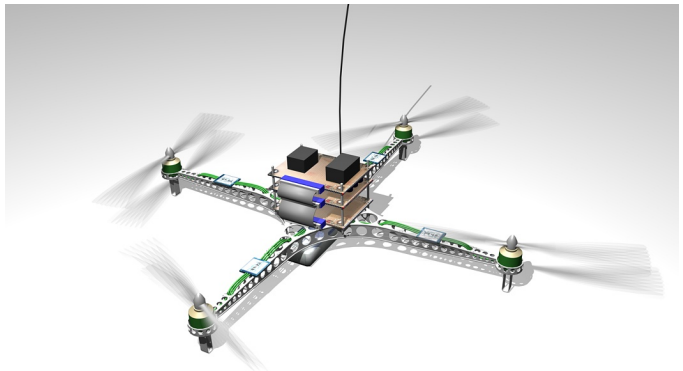


- Befehlsschnittstelle für Oszilloskop
- Auswertung von Benutzereingaben
  - Unterbrecher-, Hintergrundbetrieb, Periodischer Zusteller

    → **nicht SpSL implementieren!**
- *Moduswechsel* (VL 4-3)
  - Dynamische Anpassung je nach Situation (Rekonfiguration der Ablauftabellen)
  - Systemweite Koordination mittels *Zustandsmaschine*
- Erweiterte Übung
  - Rangfolge
  - Mailboxen, Events

Übersicht

Beispiel-Zustandsautomat



- I4Copter grundsätzlich instabil

  ⇝ Fluglageregelung zwingend erforderlich

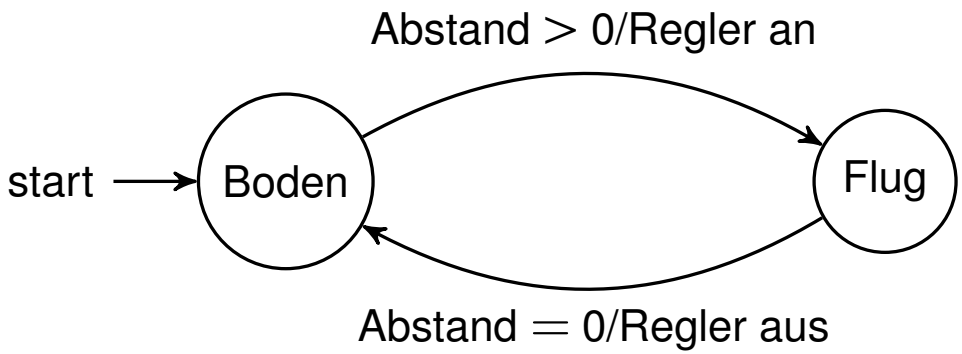
- Im Flug: Regelkreis geschlossen
- Aber: Am Boden Regelkreis offen

  ⇝ Regler darf am Boden nicht laufen

  - Andernfalls Verfälschung des Reglerzustands

⇒ Zustandsmaschine mit zwei Zuständen

Zustandsautomat



Datenstrukturen

```
1  enum FlightState {
2    Landed,
3    InFlight
4  };
5
6  enum Event {
7    GroundDistanceGreaterThanZero,
8    GroundDistanceZero
9  };
10
11 static FlightState g_flightState;
```

Ereignisbehandlung

```
1  static void state_init(void) {
2    calibrateSensors();
3    initializeController();
4
5    g_flightState = Landed;
6  }
```

```
1  static void event_loop(void) {
2    state_init();
3    while (true) {
4      Event event = waitForEvent();
5      state_transition(event);
6    }
7  }
```

- In Zustand z.B. zyklischer Ablaufplan
- Analyse einzelner Zustände

Zustandsübergang

```
1  static void state_transition(Event event) {
2    switch (g_flightState) {
3      case Landed:
4      state_transition_landed(event);
5      break;
6      case InFlight:
7      state_transition_inFlight(event);
8      break;
9    }
10 }
11 static void state_transition_landed(Event event) {
12   if (event == GroundDistanceGreaterThanZero) {
13     action_controllerOn();
14     g_flightState = InFlight;
15   }
16 }
17 static void state_transition_inFlight(Event event) {
18   if (event == GroundDistanceZero) {
19     action_controllerOff();
20     g_flightState = Landed;
21   }
22 }
```

Literatur

[1] John Regehr.
Safe and structured use of interrupts in real-time and embedded software.
Handbook of Real-Time and Embedded Systems, I. Lee, JY-T. Leug, and SH Son, Eds. Chapman and Hall/CRC, pages 1–13, 2007.

[2] John Regehr and Usit Duongsaa.
Preventing interrupt overload.
In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '05)*, pages 50–58, New York, NY, USA, 2005. ACM Press.