

# Echtzeitsysteme

Übungen zur Vorlesung

Cyclic 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



- 1 Wiederholung: Cyclic Executive
- 2 Vorgriff: Nicht-periodische Aufgaben
- 3 Implementierung: Cyclic Executive
- 4 Hinweis zur Aufgabe 5



- 1** Wiederholung: Cyclic Executive
- 2 Vorgriff: Nicht-periodische Aufgaben
- 3 Implementierung: Cyclic Executive
- 4 Hinweis zur Aufgabe 5



## Herkömmliche Zeitsteuerung

- potentiell komplexe Ablaufpläne
  - ↪ eventuell inklusive Verdrängung
- Einplanung zu jedem Takt möglich
- Terminüberprüfung für jeden Task

## Implementierung ohne regelmäßigen Zeitgeber (tickless)

- + weniger Zeitgeberereignisse
- Einplanung/Terminüberprüfung weiterhin zu jedem Takt möglich

## Strukturierte Ablaufplanung

- Einführung künstlicher, zeitlicher Struktur
- weniger Zeitpunkte für Einplanung/Terminüberprüfung





# Vor-/Nachteile zyklischer Ablaufpläne



## Zyklisches Ablaufmodell liefert wohlgeordnete Ablaufpläne

- Eine feste Rahmengröße mit definierten Schranken
- Ablaufplanung (↔ Zuteilung Aufträge zu Rahmen) findet **offline** statt
- **Einlastung** und **Terminüberwachung** zu definierten Zeitpunkten

## - **Busy-Loop-Verhalten** innerhalb eines Rahmens (vgl. IV-2/12)

- Sequentielle, kooperative Abarbeitung der Aufträge
- Keine individuelle **Laufzeitüberwachung** und **Ausnahmebehandlung**
- Anfällig für Jitter und mangelnde Periodizität

## + **Niedrige Verwaltungsgemeinkosten**

- **Einlastung** und **Terminüberwachung** findet nur an den Rahmengrenzen statt
- Keine Verdrängung (engl. *preemption*) (vgl. III-2/13)
- Minimalistisches Laufzeitsystem (Dispatcher+Terminprüfung genügt)

## + **Hohe Vorhersagbarkeit**

- Einziger Interrupt ist der Zeitgeber an den Rahmengrenzen
- **Unterbrechungsfreier Durchlauf** innerhalb der Rahmen
- Vereinfacht die WCET-Analyse ungemein (vgl. Kapitel III-3)



# Randbedingungen für die Rahmenlänge

Lang genug und so kurz wie möglich halten...

## Terminüberwachung unterstützen $\leadsto f$ hinreichend kurz

- 1 Erfordert eine rechtzeitige Auslösung:  $f \leq p_i$ , für alle  $1 \leq i \leq n$
- 2 Möglich unter der Bedingung:  $2f - ggT(p_i, f) \leq D_i$ , für alle  $1 \leq i \leq n$
- 3  $f$  teilt die Hyperperiode  $H$  so, dass gilt:  $\lfloor p_i/f \rfloor - p_i/f = 0$ , für ein  $i$  mit  $1 \leq i \leq n$

## Jobverdrängung vermeiden $\leadsto f$ hinreichend lang

- 4 Erfüllt, wenn gilt:  $f \geq \max(e_i^f)$ , für  $1 \leq i \leq H/f$ 
  - $e_i^f$  gibt die WCET aller Aufträge im Rahmen  $i$  an
  - Jeder Auftrag läuft in der durch  $f$  gegebenen Zeitspanne komplett durch
  - Erste Abschätzung nach unten:  $f \geq \max(e_i)$ , für  $1 \leq i \leq n$

**⚠ Ermittlung von  $\max(e_i^f)$  erfolgt nachgelagert:**

- Kann erst durch konkrete Ablaufplanung beantwortet werden
- Iterativer Prozess  $\mapsto$  Wiederholung für jedes potentielle  $f$



Aufgabe $T_i$	<u>Periode <math>p_i</math></u> ms	<u>WCET <math>e_i</math></u> ms	<u>Termin <math>D_i</math></u> ms
$T_1$	9	2	5
$T_2$	18	3	8
$T_3$	45	3	45



- 1 Wiederholung: Cyclic Executive
- 2 Vorgriff: Nicht-periodische Aufgaben**
- 3 Implementierung: Cyclic Executive
- 4 Hinweis zur Aufgabe 5





### Nicht-periodische Aufgaben

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

### Nicht-periodische Arbeitsaufträge

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



### Basistechniken zur Umsetzung

- **Unterbrecherbetrieb**  $\leadsto$  bevorzugt nicht-periodische Aufgaben
- **Hintergrundbetrieb**  $\leadsto$  stellt nicht-periodische Aufgaben hinten an
- **Zusteller**  $\leadsto$  konvertiert nicht-period. in periodische Aufgaben
  - Spezielle periodische Aufgabe  $T_s = (p_s, e_s)$
  - Ausführungsbudget, Auffüllperiode und -regeln
  - Abbildung auf Prioritätswarteschlange (z. B. AJQ)

### Slack Stealing

- Idee: Termin ist maßgeblich  
 $\leadsto$  *Verschieben* periodischer Aufgaben möglich
- *Erfordert Unterbrecherbetrieb*
- Problem: **Schlupfzeit** bestimmen
  - Zeitsteuerung (mit Rahmen): Einfach  $\leadsto f - x_k$
  - Ereignissteuerung: schwierig  $\leadsto$  dynamischen Berechnung



### 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



- 1 Wiederholung: Cyclic Executive
- 2 Vorriff: Nicht-periodische Aufgaben
- 3 Implementierung: Cyclic Executive**
- 4 Hinweis zur Aufgabe 5



# Busy Loop

```
1 void main(void) {  
2     while (true) {  
3         Task0();  
4         Task1();  
5         Task2();  
6         Task3();  
7     }  
8 }
```

## Vorteile:

- Geringe Verwaltungsallgemeinkosten
- Simpel, übersichtlich, ...

## Nachteile:

- Nur *eine Periode*, *keine Deadline-Überprüfung* möglich
- Mathematische *Analyse unmöglich*



# Multi-Perioden-Hauptschleife

Anforderung: wir wollen unterschiedliche Perioden haben

Lösung:

- Jede Aufgabe hat ein *Aktivierungs-Flag*
- Feste Abarbeitungsreihenfolge innerhalb eines Durchlaufs

## Multiraten-Hauptschleife

```
1 void main(void) {  
2     while (true) {  
3         wait_for_timer_tick();  
4         if (activated0) { activated0 = false; Task0(); }  
5         if (activated1) { activated1 = false; Task1(); }  
6         if (activated2) { activated2 = false; Task2(); }  
7         if (activated3) { activated3 = false; Task3(); }  
8     }  
9 }
```

Setzen der Flags in der Hauptschleife problematisch

→ Lang laufender Task kann Flag-Setzen/*Deadlineüberprüfung* verzögern



## Setzen der Flags in der Hauptschleife problematisch

↪ Lang laufender Task kann Flag-Setzen/*Deadlineüberprüfung* verzögern

## Lösung: Setzen der Flags in Zeitgeber-Interruptbehandlung

```
1 uint8_t timer = 0;
2 ...
3 // ISR:
4 ++timer; // Interrupt alle 1ms
5 ...
6 if ((timer % 5) == 0) { activated0 = true; } // Task0 alle 5ms
7 if ((timer % 10) == 0) { activated1 = true; } // Task1 alle 10ms
8 if ((timer % 20) == 0) { activated2 = true; } // Task2 alle 20ms
9 if ((timer % 100) == 0) { activated3 = true; } // Task3 alle 100ms
10
11 if (timer >= 100) { timer = 0; } // Ueberlaufbehandlung
```



## Einschub: Schlüsselwort volatile

- Bei einem Interrupt wird `timer_event = 1` gesetzt
- Aktive Warteschleife wartet, bis `timer_event != 0`
- Flag (scheinbar) in Schleife nicht verändert  $\leadsto$  Compiler-Optimierung
  - `timer_event` wird einmalig vor der Warteschleife in Register geladen
    - ☞ Endlosschleife
- `volatile` erzwingt das Laden bei jedem Lesezugriff

```
1 static uint8_t timer_event = 0;
2 ISR (INT0_vect) { timer_event = 1; }
3
4 void main(void) {
5     while(1) {
6         while(timer_event == 0) { /* warte auf Timer-Event */ }
7         /* bearbeite Timer-Event */
1 volatile static uint8_t timer_event = 0;
2 ISR (INT0_vect) { timer_event = 1; }
3
4 void main(void) {
5     while(1) {
6         while(timer_event == 0) { /* warte auf Timer-Event */ }
7         /* bearbeite Timer-Event */
```





## Setzen der Flags in der Hauptschleife problematisch

~> Lang laufender Task kann Flag-Setzen/*Deadlineüberprüfung* verzögern

## Lösung: Setzen der Flags in Zeitgeber-Interruptbehandlung

```
1 volatile uint8_t timer = 0;
2 ...
3 // ISR:
4 ++timer; // Interrupt alle 1ms
5 ...
6 if ((timer % 5) == 0) { activated0 = true; } // Task0 alle 5ms
7 if ((timer % 10) == 0) { activated1 = true; } // Task1 alle 10ms
8 if ((timer % 20) == 0) { activated2 = true; } // Task2 alle 20ms
9 if ((timer % 100) == 0) { activated3 = true; } // Task3 alle 100ms
10
11 if (timer >= 100) { timer = 0; } // Ueberlaufbehandlung
```



- Tastendruckzähler: Zählt mittels Variable zaehler
  - Inkrementierung in der Unterbrechungsbehandlung
  - Dekrementierung im Hauptprogramm zum Start der Verarbeitung

## Hauptprogramm H

```
1 ; volatile uint8_t zaehler;  
2 ; C-Anweisung: zaehler--;  
3 lds r24, zaehler  
4 dec r24  
5 sts zaehler, r24
```

## Interruptbehandlung I

```
1 ; C-Anweisung: zaehler++  
2 lds r25, zaehler  
3 inc r25  
4 sts zaehler, r25
```

Zeile	zaehler	r24	r25
-	5		
3 H	5	5	-
4 H	5	4	-
2 I	5	4	5
3 I	5	4	6
4 I	6	4	6
5 H	4	4	-



## Vorteile

- Einfach, übersichtlich, wenige Ressourcen notwendig, ...
- Mehrere Perioden, Deadlineüberprüfung, erleichtert WCET-Analyse
- Mathematische Analyse anwendbar

## Probleme der Implementierung: *Nebenläufige Zugriffe*

(Sichtbarkeits-)Synchronisation:

- 1 zwischen Zeitgeberunterbrechung und main-if/else
- 2 beim Setzen der Flags

## Andere Namen in der Literatur:

Main Loop Scheduling, Main Loop Tasker, Prioritized Cooperative Multitasker, Non-preemptive Scheduler, ...



- 1 Wiederholung: Cyclic Executive
- 2 Vorgriff: Nicht-periodische Aufgaben
- 3 Implementierung: Cyclic Executive
- 4 Hinweis zur Aufgabe 5**



## Wichtige Hinweise

Basisübung: Reine Textaufgabe, *Denksportaufgabe*

~> keine Implementierung notwendig

- Kern der Aufgabe: Auswirkung der Rahmenlänge

Erweiterte Übung: Implementierung einer *Cyclic Executive*

- Überprüfung der Lauffähigkeit und Deadlines von Jobs
- Vereinfachte Ausnahmebehandlung:  
Ausgabe welcher Task Deadline überschritten hat
- Verwendung *eines* eCos Alarms

