

Cyclic Scope

Strukturelemente in Echtzeitsystemen

Tobias Klaus Florian Schmaus Peter Wägemann

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

12. Dezember 2016



Hinweise: Simple Scope

- Neue Vorgabe verfügbar
 - app.c
 - Sensorfunktionen verfügbar
- Hyperperiode fest auf 100 ms kodiert
- Wenn längere Zeit kein Plot der Ergebnisse
 - 256 Scheduling-Entscheidungen bis Plot angezeigt wird
 - event-triggered: ms_to_cyg_ticks, ms_to_ezs_ticks
 - time-triggered: ms_to_ticks
- Zu langes Sampling: Überlauf des Timers



Übersicht

- 1 Hinweise: Simple Scope
- 2 Wiederholung: Cyclic Executive
- 3 Implementierung: Cyclic Executive
- 4 Hinweis zur Aufgabe 5



Randbedingungen für die Rahmenlänge

Lang genug und so kurz wie möglich halten...

Jobverdrängung vermeiden $\leadsto f$ hinreichend lang

- 1 ist erfüllt, wenn gilt: $f \geq \max(e_i)$, für $1 \leq i \leq n$
 - \leadsto jeder Job läuft in der durch f gegebenen Zeitspanne durch
- 2 f teilt die Hyperperiode H so, dass gilt: $\lfloor p_i/f \rfloor - p_i/f = 0$
 - \leadsto ermöglicht die zyklische Ausführung des Ablaufplans

- das Intervall H heißt **großer Durchlauf** \equiv Hyperperiode
- das Intervall der Länge f heißt **kleinster Durchlauf**

Terminüberwachung unterstützen $\leadsto f$ hinreichend kurz

- 3 erfordert eine rechtzeitige Auslösung: $f \leq p_i$, für $1 \leq i \leq n$
- 4 ist möglich unter der Bedingung: $2f - \text{ggT}(p_i, f) \leq D_i$

- anstehenden Aufgaben „passend“ auf die Rahmen verteilen
 - \leadsto Jobs zwischen Auslösezeit und Termin erledigen



Beispielsystem

Aufgabe T_i	Periode p_i ms	WCET e_i ms	Termin D_i ms
T_1	9	2	5
T_2	18	3	8
T_3	45	3	45



Übersicht

- 1 Hinweise: Simple Scope
- 2 Wiederholung: Cyclic Executive
- 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*
- In jedem Schleifendurchlauf Ausführung *höchstens* eines Tasks
- Abbildung der Prioritäten durch Abarbeitungsreihenfolge

Multiraten-Hauptschleife

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



Multi-Perioden-Zeitgeber

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 ++timer; // Interrupt alle 1ms
4 ...
5 if ((timer % 5) == 0) { activated0 = true; } // Task0 alle 5ms
6 if ((timer % 10) == 0) { activated1 = true; } // Task1 alle 10ms
7 if ((timer % 20) == 0) { activated2 = true; } // Task2 alle 20ms
8 if ((timer % 100) == 0) { activated3 = true; } // Task3 alle 100ms
9
9 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 ~> 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
3 void main(void) {
4 while(1) {
5     while(timer_event == 0) { /* warte auf Timer-Event */
6         /* bearbeite Timer-Event */
    
```

```

1 volatile static uint8_t event = 0;
2 ISR (INT0_vect) { event = 1; }
    
```



```

3 void main(void) {
4 while(1) {
5     while(event == 0) { /* warte auf Event */
6         /* bearbeite Event */
    
```

Einschub: Lost-Update-Problematik

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

Hauptprogramm H

Interruptbehandlung I

```

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



Fazit Cyclic Executive

Vorteile

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

Probleme der Implementierung: *Wettkampfsituationen*

- 1 zwischen Zeitgeberunterbrechung und main-if/else
~> *Prioritätsverletzung* (fällt praktisch jedoch nicht ins Gewicht)
- 2 beim Setzen der Flags ~> ggf. Unterbrechungen abschalten

Andere Namen in der Literatur:

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



- 1 Hinweise: Simple Scope
- 2 Wiederholung: Cyclic Executive
- 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

