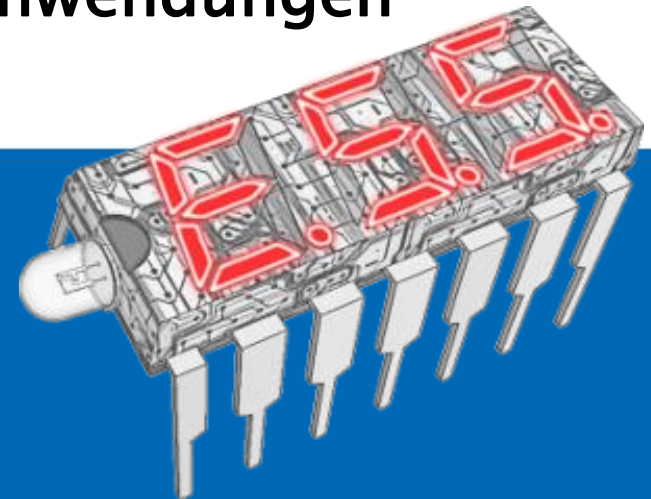
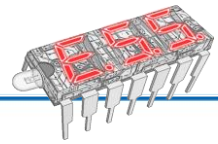


Prinzipien und Komponenten Eingebetteter Systeme (PKES)

(10) Scheduling in eingebetteten Anwendungen

Sebastian Zug
Arbeitsgruppe: Embedded Smart Systems





Praktische Aufgabe 5

- Alternative A:

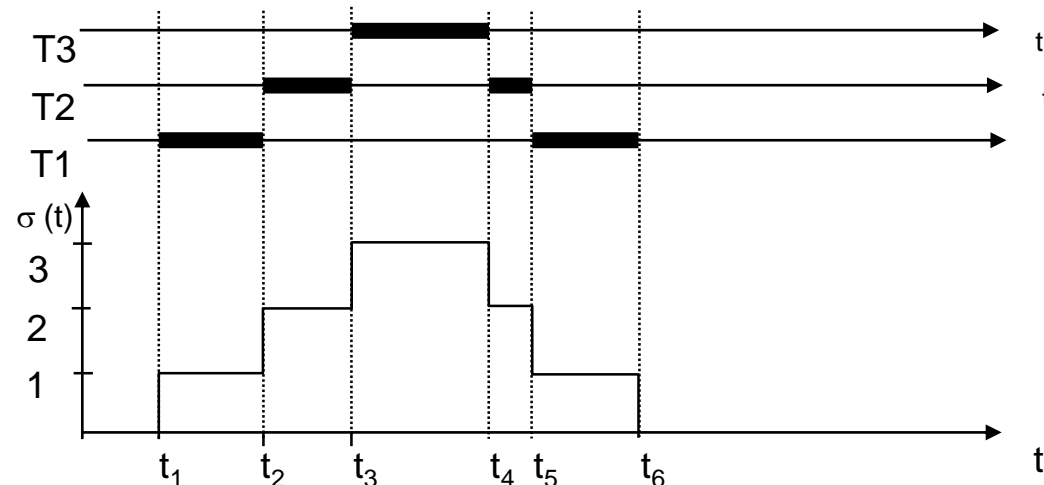
Entwickeln Sie eine Anwendung, die es dem Nutzer erlaubt, auf seinem Rechner einen Roboterbewegung zu definieren

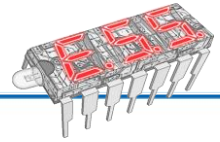
Geradeaus 30cm, Linksschwenk 90 Grad, Geradeaus 30cm ... etc.

Nachdem die Eingabe abgeschlossen ist, wird das Bewegungsmuster ausgeführt.

- Alternative B (Fortgeschrittene):

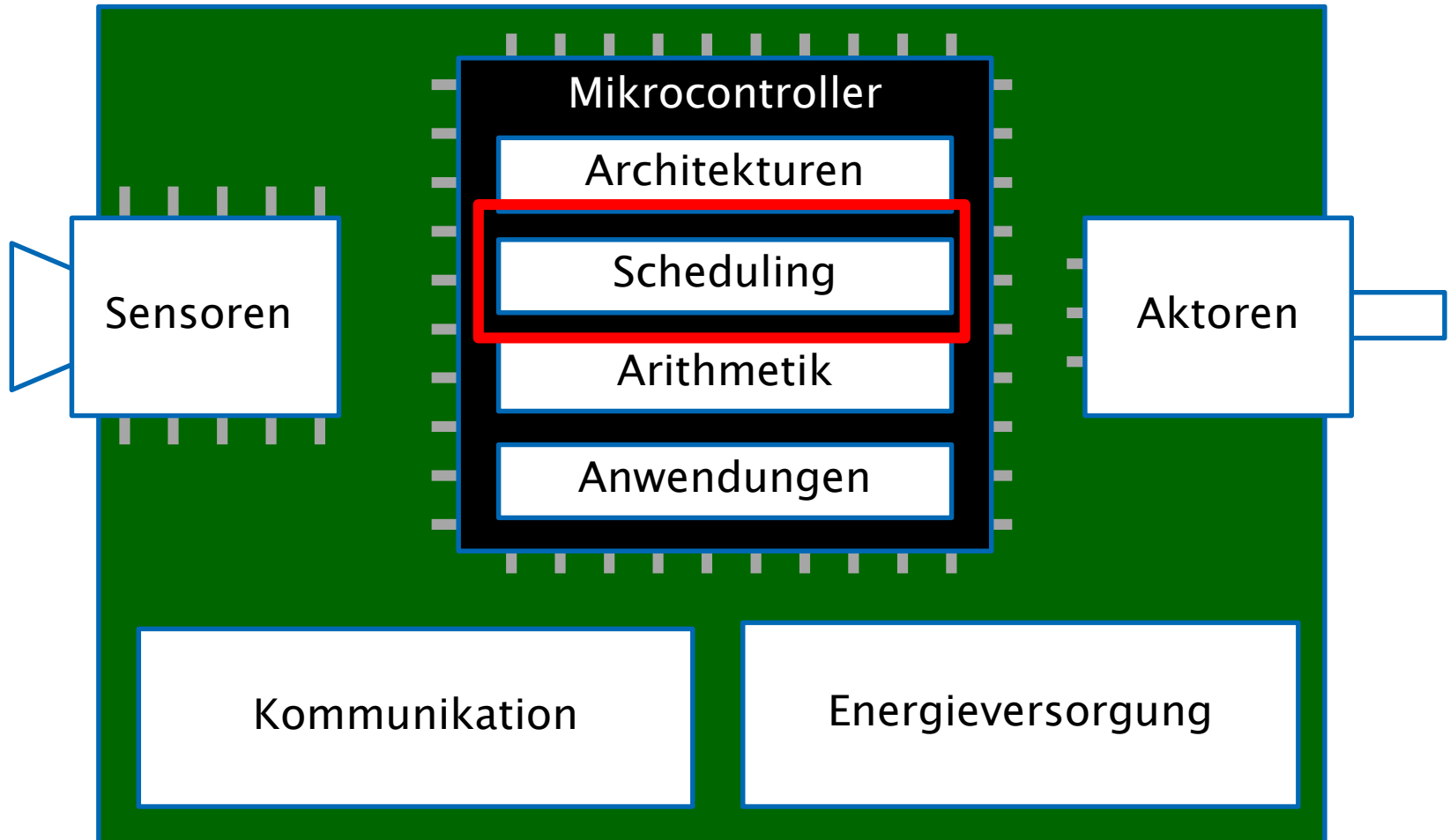
Erweitern Sie das in der Vorlesung gezeigte FreeRTOS Beispiel um eine (leichtgewichtige) Funktion, die es erlaubt, die Aufrufhierarchie abzubilden.

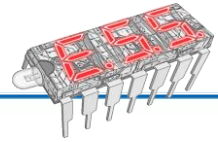




„Veranstaltungslandkarte“

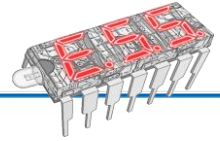
Fehlertoleranz, Softwareentwicklung





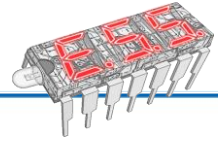
Fragestellungen dieser Vorlesung

1. Welche eingebetteten Applikationen sind auf harte Echtzeitbedingungen angewiesen?
2. Welches Scheduling Kriterium gilt für harte Echtzeitbedingungen?
3. Erklären Sie die Task-Zustandsmaschine, wie sie in der Vorlesung vorgestellt wurde.
4. Wie wird die Abhängigkeit zwischen Tasks im EDF Verfahren berücksichtigt?
5. Welche Erweiterung bringt der Least Leaxity Ansatz gegenüber dem EDF mit?
6. Worin unterscheidet sich das Rate-Monotonic Scheduling von den anderen Verfahren? Welches ist die wichtigste Einschränkung?
7. Worin liegen die Vorteile beim Einsatz eines RTOS? Welche Funktion sollte dies abdecken?



Literaturhinweise

- Peter Marwedel
Eingebettete Systeme
Springer Lehrbuch, 2008
- Dieter Zöbel
Echtzeitsysteme – Grundlagen der Planung
Springer Lehrbuch, 2008



Problemstellung – Anwendung

Parallel laufende Aufgaben in eines Controllers für ein inverses Pendel

1. Regelung eines Motors	5ms
2. Empfang von Userinputs	20ms
3. Überwachung der Spannungsversorgung	100ms
4. Kommunikation des Zustandes	1000ms

Dauer der einzelnen Tasks

2. Empfang:

$9600 \text{ Baud} = 1 \text{ s} / 9600 = 104 \mu\text{s pro Bit}$

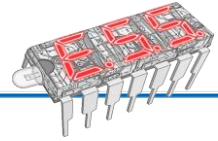
mit $4 \times 8 \text{ bit} + 4 \times \text{Startbit} + 4 \times \text{Stop} = 40 \text{ Bit}$

Gesamtdauer $40 \times 104 \mu\text{s} = 4.16 \text{ ms}$

3. Überwachung der Spannungsversorgung

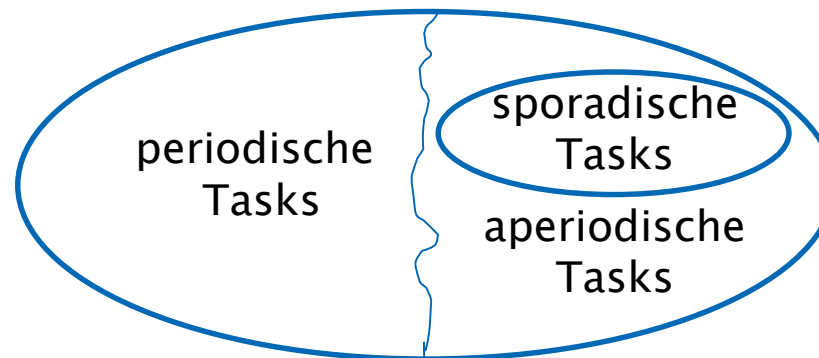
Aufruf eines unterprogrammes, Analoge Messung,
Mittelwertbildung

Bestimmung über Simulator



Charakterisierung von Tasks – Zeitverhalten

- Periodische Tasks ... werden mit einer bestimmten Frequenz f regelmäßig aktiviert.
 - Durchlaufen einer Regelschleife
 - Pollendes Abfragen eines Sensors
- Aperiodische Tasks ... lassen sich nicht auf ein zeitlich wiederkehrendes Muster abbilden.
 - Tastendruck auf einem Bedienfeld
- Sporadische Tasks ... treten nicht regulär auf. Man nimmt aber eine obere Schranke bzgl. der Häufigkeit ihres Aufrufs an.
 - Fahrradacho (obere Schranke = Geschwindigkeit)



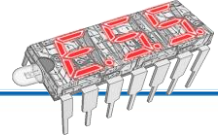


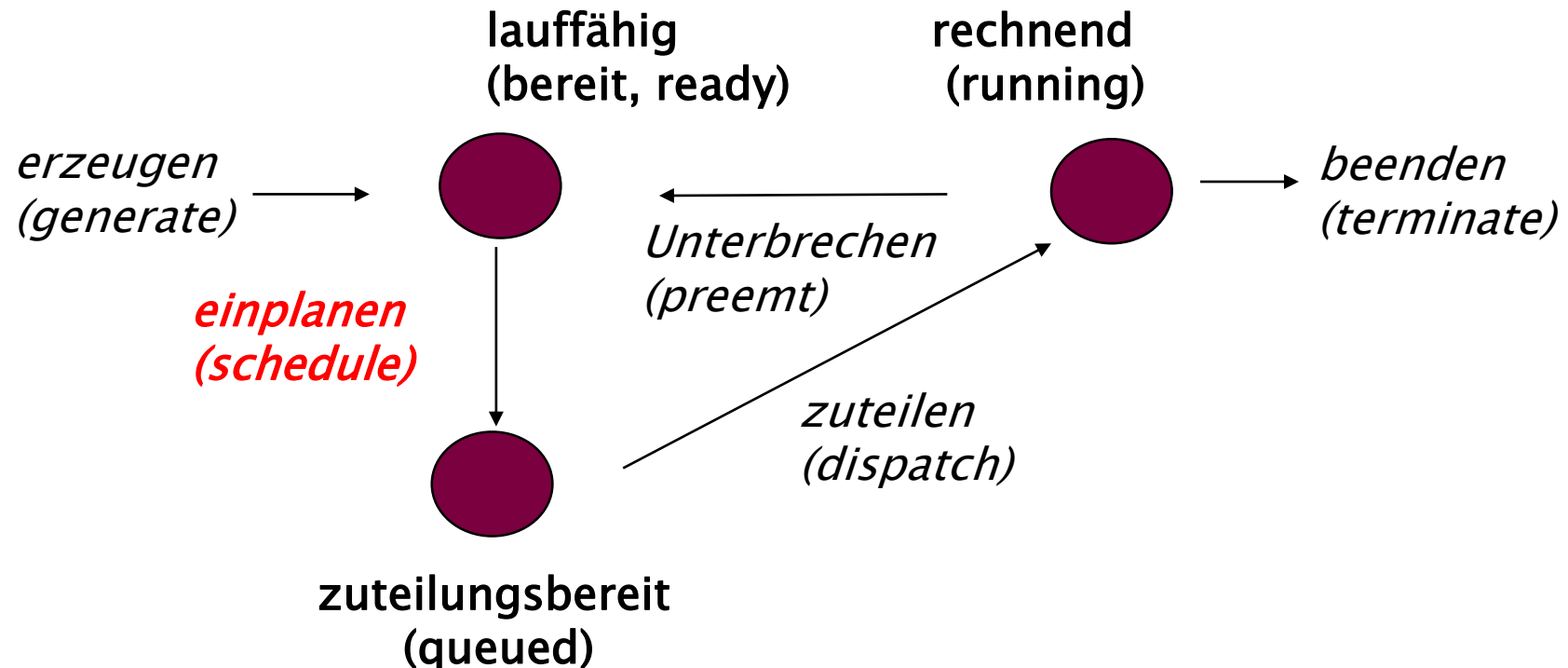
Abbildung der Tasks auf einen Automaten

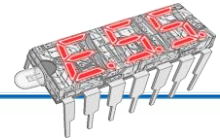
1. Einplanbarkeitsanalyse
(*feasibility check*)
2. Planerstellung
(*schedule construction*)
3. Prozessorzuteilung
(*dispatching*)

Strategisches Scheduling

Operatives Scheduling

Dispatching





Echtzeit Kostenfunktionen

Maximale Zahl verspäteter Tasks:

$$N_{late} = \sum_{(i=1, \dots, n)} miss(c_i)$$

mit:

$$miss(c_i) = \begin{cases} 0 & \text{if } c_i \leq d_i \\ 1 & \text{sonst} \end{cases}$$

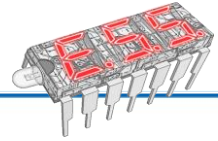
Maximale Verspätung:
(maximum lateness)

$$L_{max} = \max_i (c_i - d_i)$$

Für Prozesse mit harten Zeitbedingungen gilt dabei:

$$L_{max} \leq 0$$

Die vorgegebene Deadline muss immer eingehalten werden.



Zusammenfassung der Taskparameter

Anforderung
der Anwendung

harte Echtzeit

weiche Echtzeit

synchron bereit

asynchron bereit

Taskmodell

periodisch

aperiodisch

sporadisch

präemptiv

nicht-präemptiv

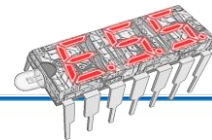
unabhängig

abhängig

Scheduler

statisch

dynamisch



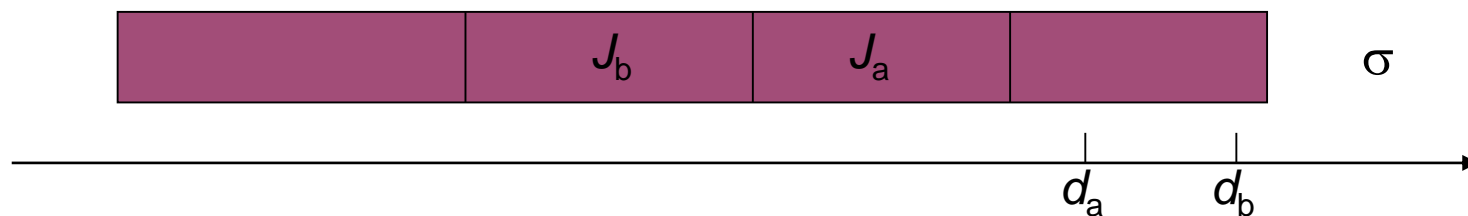
Earliest Due Date

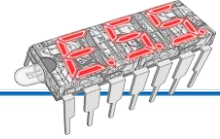
Scheduling auf **einem** Prozessor. Alle n Tasks sind unabhängig voneinander und können zur gleichen Zeit begonnen werden (zum Zeitpunkt 0).

EDD: Earliest Due Date (Jackson, 1955)

Jeder Algorithmus, der die Tasks in der Reihenfolge nicht abnehmender Deadlines ausführt, ist optimal bzgl. der Minimierung der maximalen Verspätung.

Beweis nach (Buttazzo, 2002): Sei A ein Algorithmus, der verschieden von EDD ist. Dann gibt es zwei Tasks J_a und J_b in dem von A erzeugten Schedule σ , so dass in σ J_b unmittelbar vor J_a steht, aber $d_a \leq d_b$ ist:

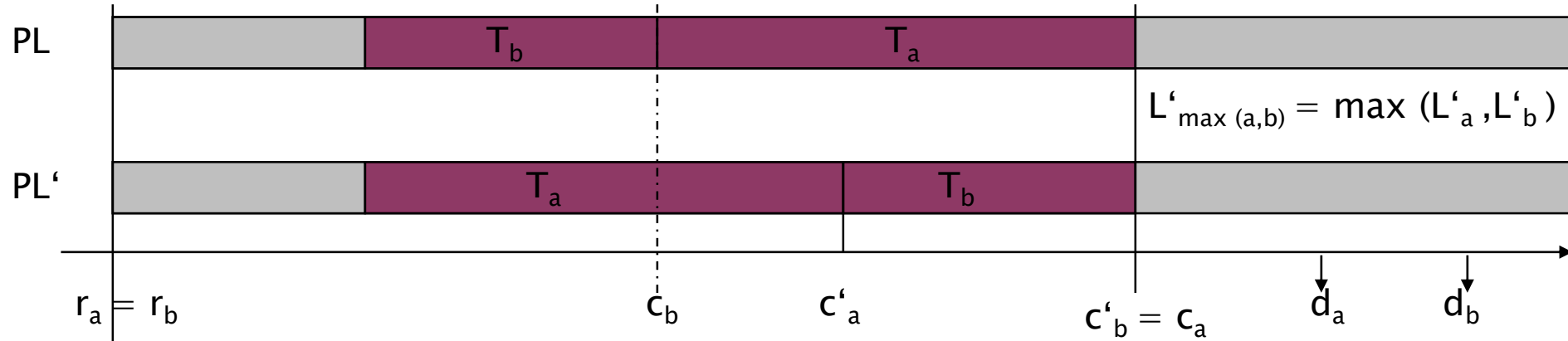




Beweis der Optimalität

L: Lateness

$$L_{\max(a,b)} = c_a - d_a$$



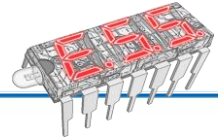
if $L'_a \geq L'_b$ then $L'_{\max(a,b)} = L'_a = c'_a - d_a < c_a - d_a$ (da gilt: $c'_a < c_a$)

if $L'_a \leq L'_b$ then $L'_{\max(a,b)} = L'_b = c'_b - d_b$
 $= c_a - d_b < c_a - d_a$ (da gilt: $d_a < d_b$)

In beiden Fällen ist $L'_{\max(a,b)} \leq L_{\max(a,b)}$!

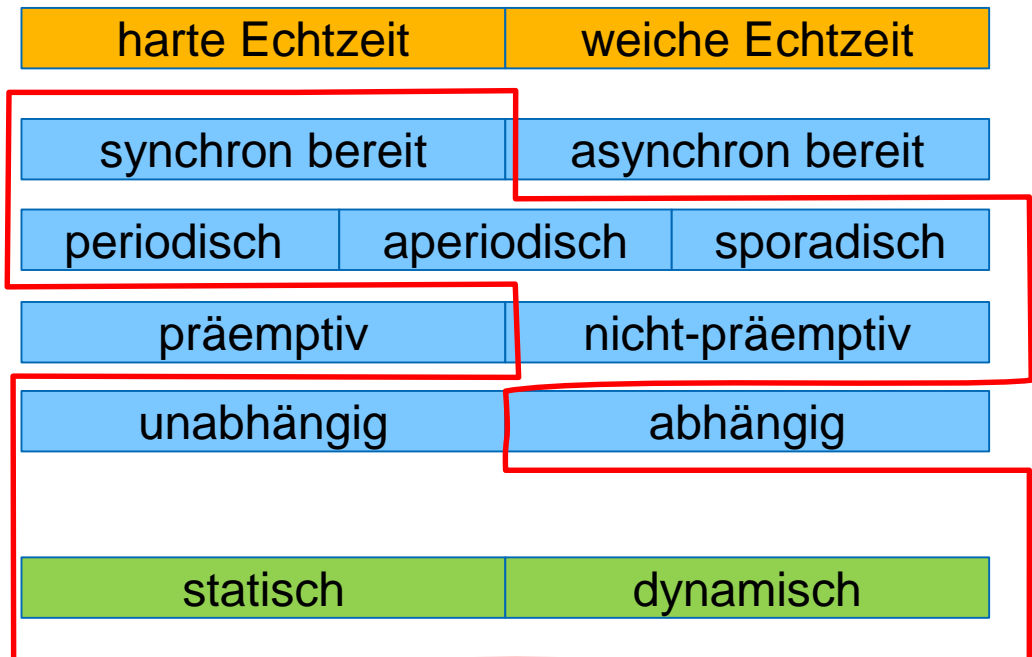
EDD generiert bei nicht unterbrechbaren Tasks einen Schedule der optimal im Hinblick auf maximale Verspätung ist.

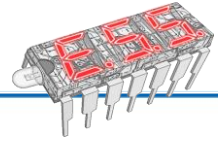
Für die Brauchbarkeit eines Plans gilt: falls EDD keinen gültigen Plan liefert, gibt es keinen !



Einschränkungen von EDD

- Starke Einschränkung des Ansatzes wegen der notwendigen gleichen Bereitzeiten



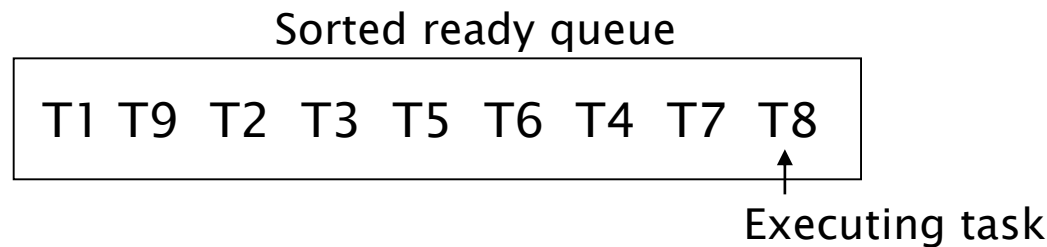


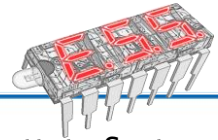
Earliest Deadline First (EDF)

[Horn, 1974]: Wenn eine Menge von n Tasks mit beliebigen Ankunftszeiten gegeben ist, so ist ein Algorithmus, der zu jedem Zeitpunkt diejenige ausführungsbereite Task mit der frühesten absoluten Deadline ausführt, optimal in Bezug auf die Minimierung der maximalen Verspätung.

Umsetzung

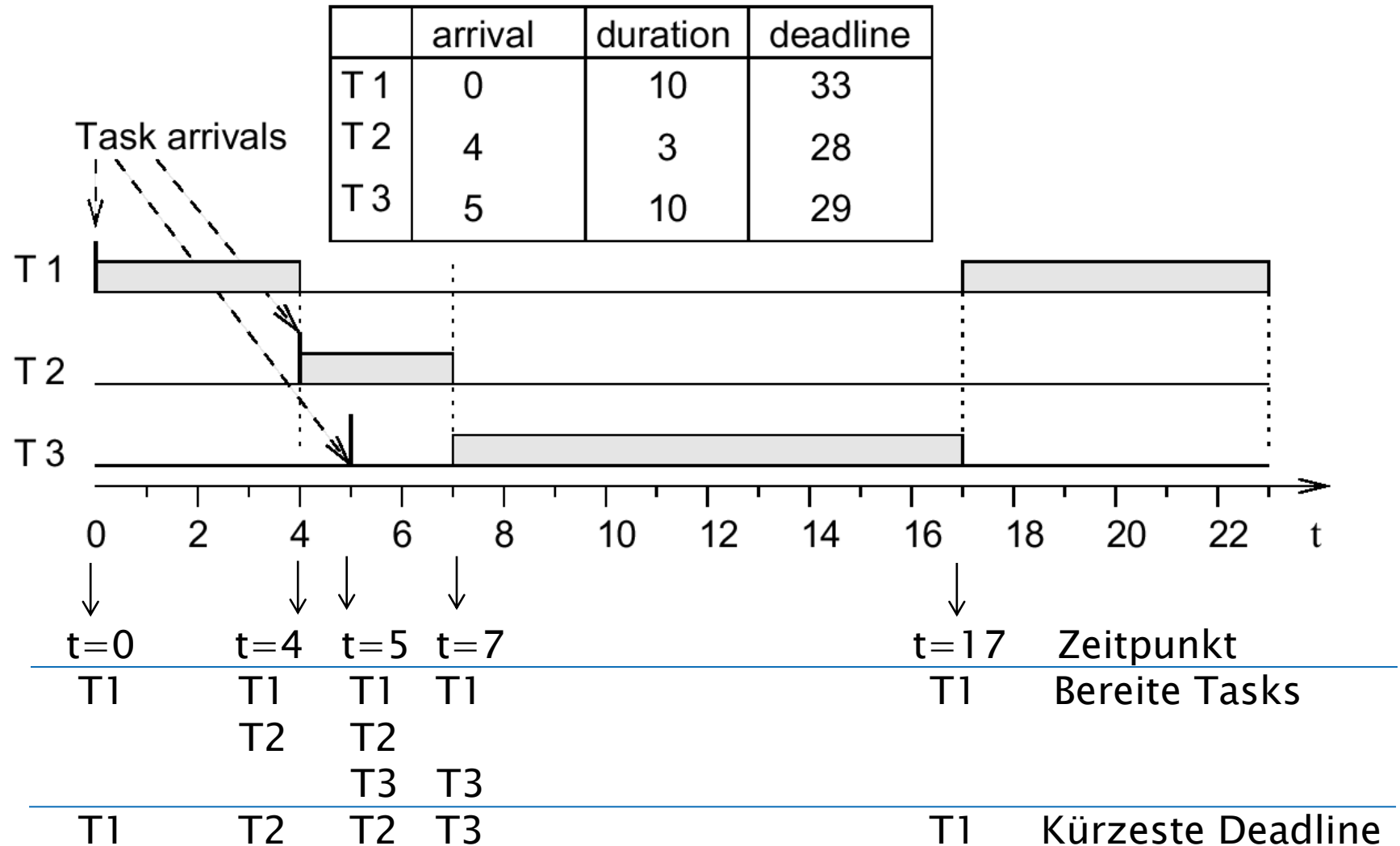
Jede ankommende ausführbare Task wird entsprechend ihrer absoluten Deadline in die Warteschleife der ausführbaren Tasks eingereiht. Wird eine neu ankommende Task als erstes Element in die Warteschlange eingefügt, muss gerade ausgeführte Task **unterbrochen** werden.

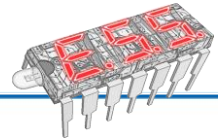




Beispiel

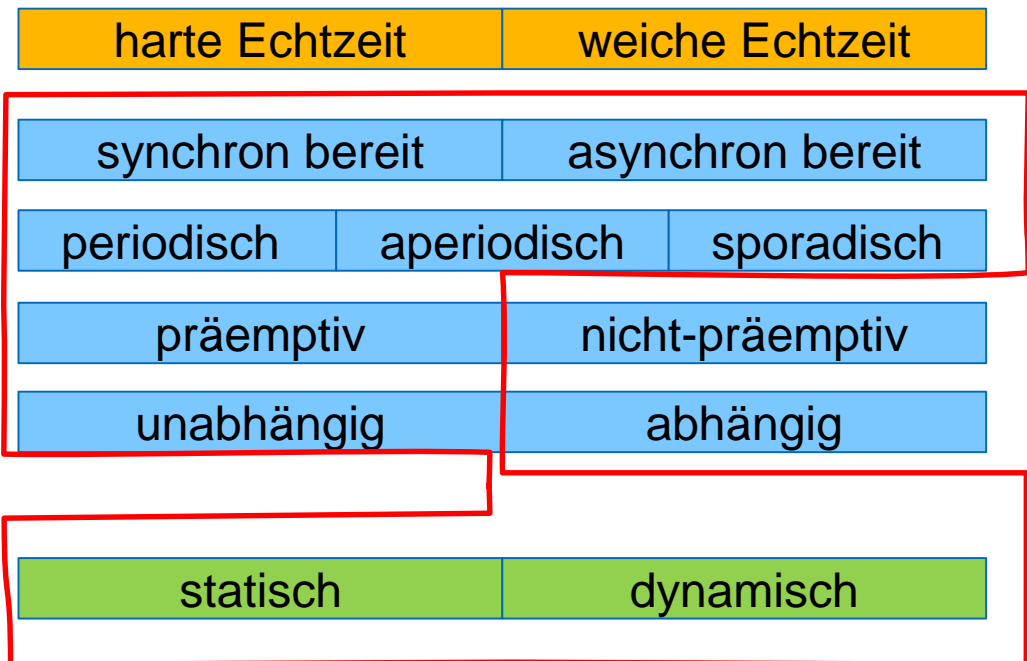
Source: Marwedel, Eingebettete Systeme

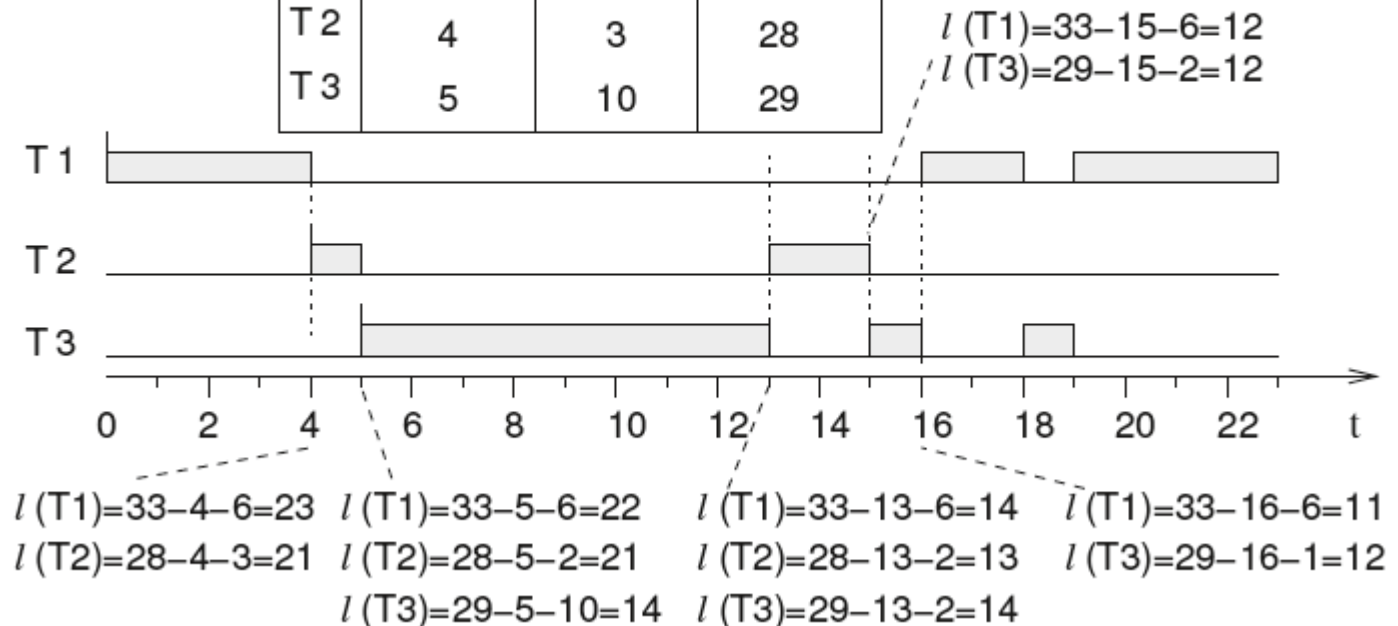


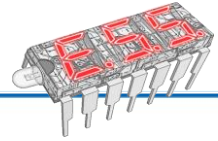


Einschränkungen von EDF

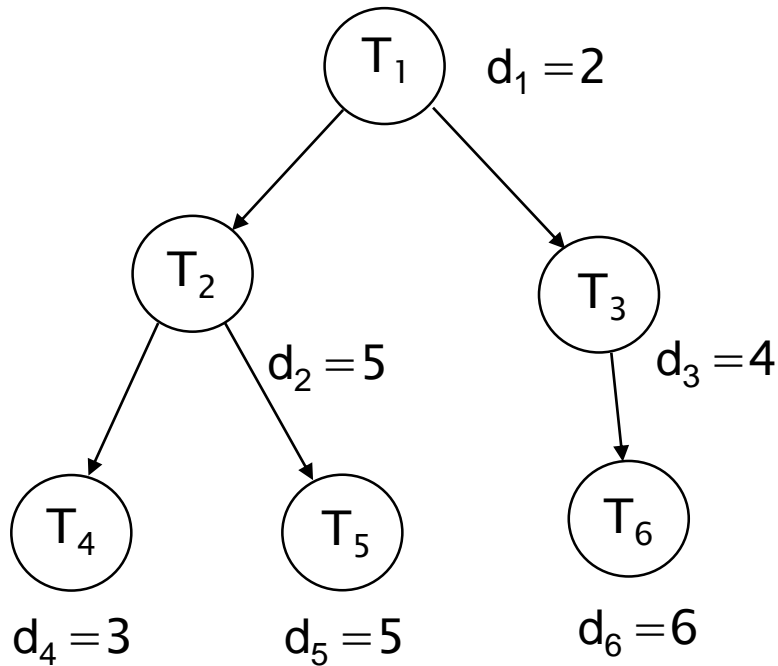
- EDF ist dabei sehr flexibel, denn es kann sowohl für präemptives, wie auch für kooperatives Multitasking verwendet werden.
- Es können Pläne für aperiodischen sowie periodischen Task entwickelt werden.
- EDF kann den Prozessor bis zur maximalen Prozessorauslastung einplanen.
- EDF ist ein optimaler Algorithmus.



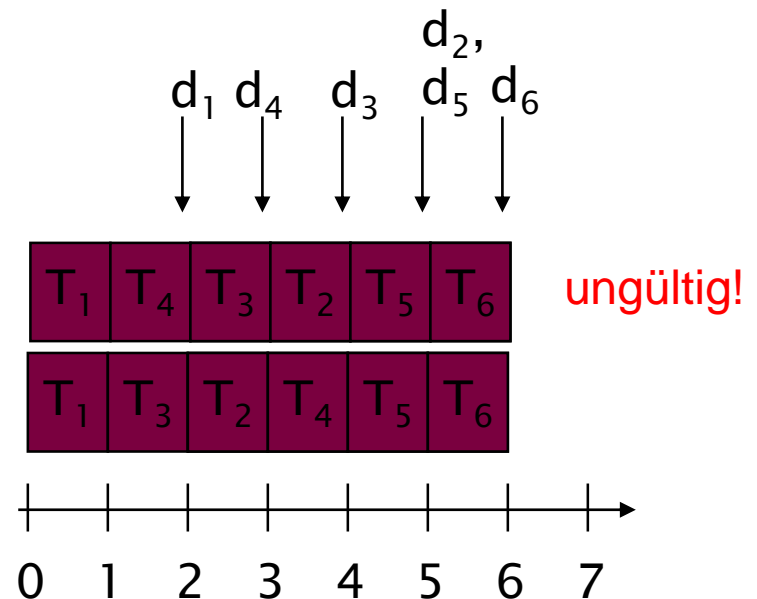


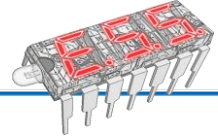


Herausforderung Abhängigkeiten

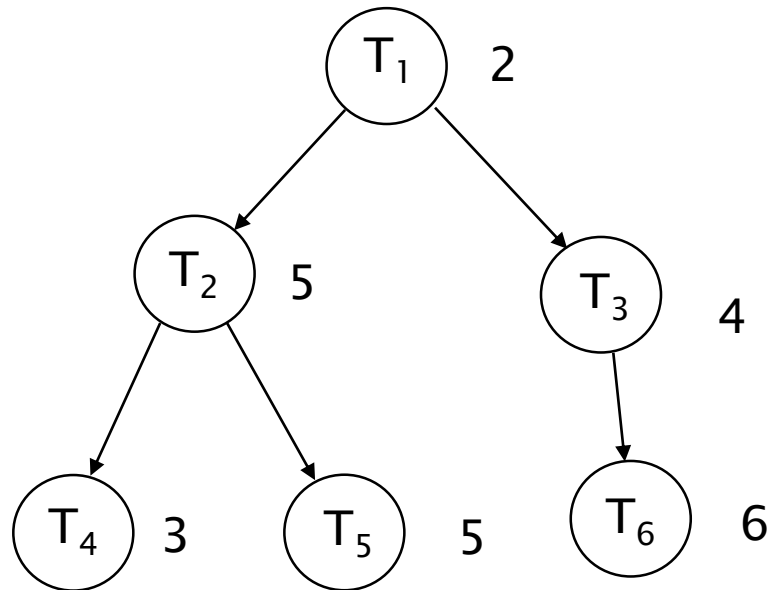


	T_1	T_2	T_3	T_4	T_5	T_6
Δe_i	1	1	1	1	1	1
d_i	2	5	4	3	5	6





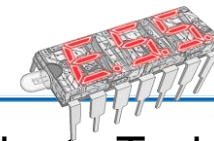
Latest Deadline First



Gegeben: Taskmenge abhängiger
Tasks $T = \{T_1, \dots, T_n\}$,
Azyklischer gerichteter
Graph, der die Vorrangrelation
beschreibt.

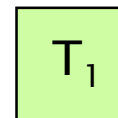
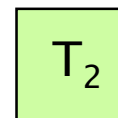
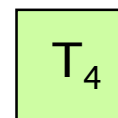
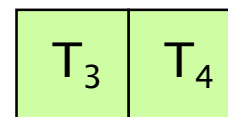
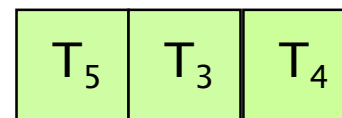
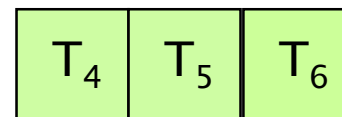
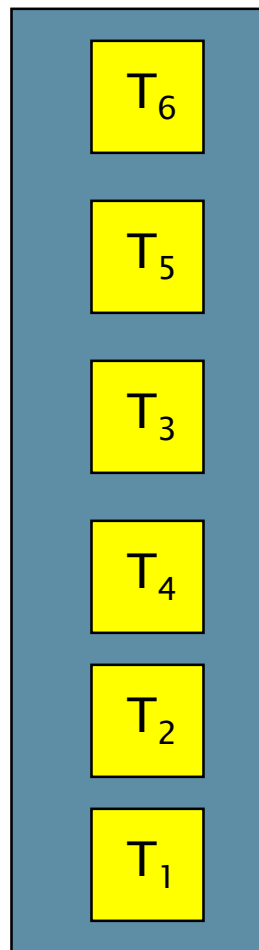
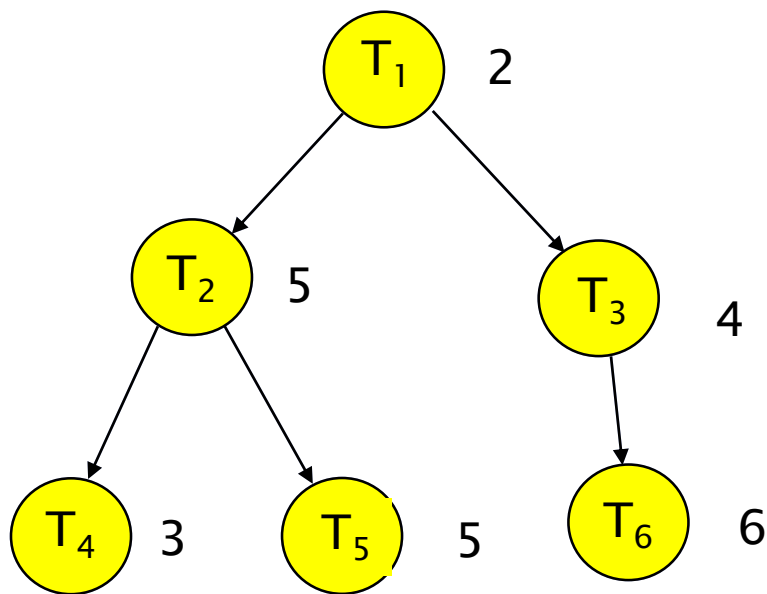
Aus der Menge der Tasks deren
Nachfolger bereits alle ausgewählt
wurden oder die keinen Nachfolger
besitzen, wählt LDF die Task mit
der spätesten Deadline aus. Die
Warteschlange der Tasks wird also
in der Reihenfolge der zuletzt
auszuführenden Tasks aufgebaut.

LDF ist ein optimaler Scheduler
(Lawler 1973)

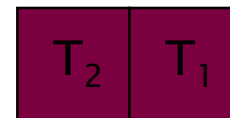
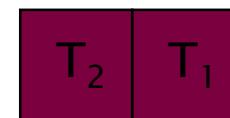
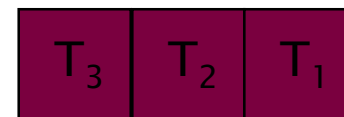


Planen nach LDF

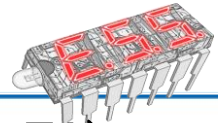
Plan Noch nicht eingeplante Tasks



Einplanbar
nach LDF



Nicht einplanbar
nach LDF



EDF mit Berücksichtigung der Vorrangrelation (EDF*)

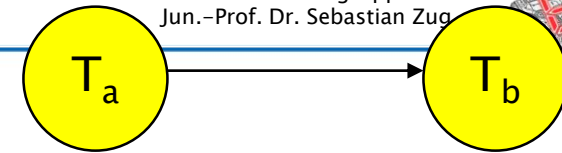
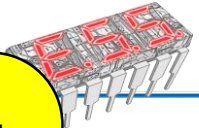
Idee: Umwandlung einer Menge abhängiger Tasks in eine Menge unabhängiger Tasks durch Modifikation der Bereitzeiten und der Deadlines.

Beobachtung:

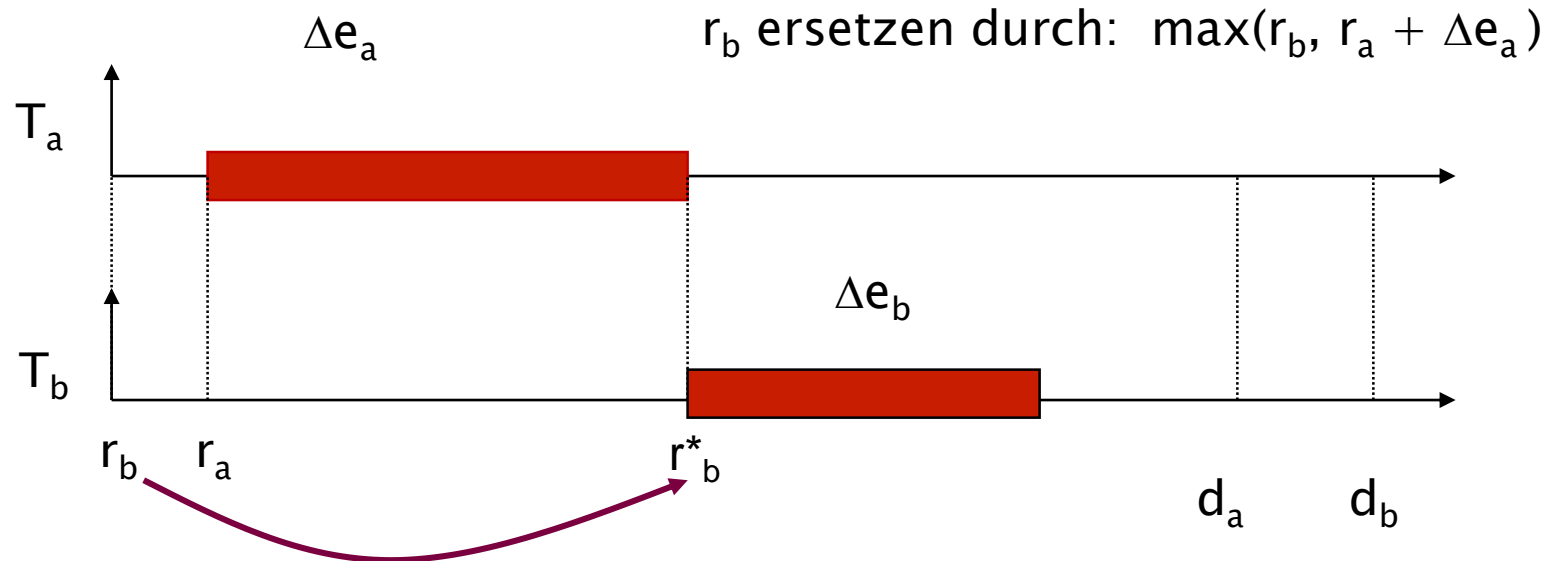
1. Eine Task kann nicht vor ihrer Bereitzeit ausgeführt werden.
2. Eine abhängige Task kann keine Bereitzeit besitzen die kleiner ist als die Bereitzeit der Task von der sie abhängt.
3. Eine Task T_b , die von einer anderen Task T_a abhängt, kann keine Deadline $d_b \leq d_a$ besitzen.

Algorithmus:

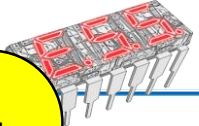
1. Modifikation der Bereitzeiten
2. Modifikation der Deadlines
3. Schedule nach EDF erstellen



Modifikation der Bereitzeiten

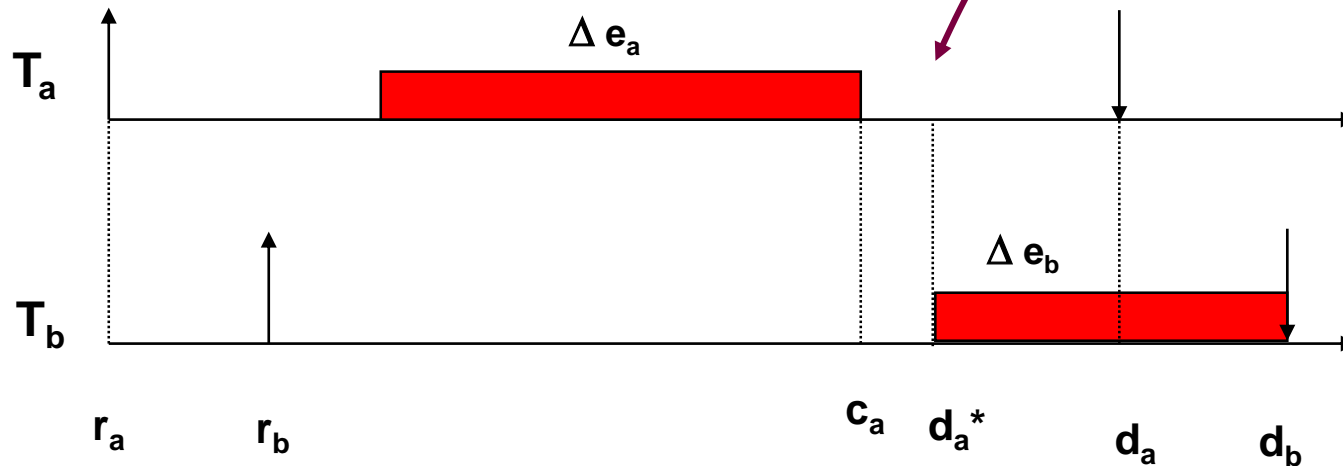


1. Für einen beliebige Anfangsknoten des Vorrang-Graphen setze $r_i^* = r_i$.
2. Wähle eine Task T_i , deren Bereitzeit (noch) nicht modifiziert wurde, aber deren Vorgänger alle modifizierte Bereitzeiten besitzen
Wenn es keine solche Task gibt: EXIT.
3. Setze $r_i^* = \max [r_i, \max(r_v^* + \Delta e_v : T_v \rightarrow T_i)]$.
4. Gehe nach Schritt 2.

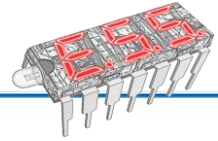


Anpassung der Deadlines

d_a ersetzen durch: $\min(d_a, d_b - \Delta e_b)$



1. Für einen beliebige Endknoten des Vorrang-Graphen setze $d_i^* = d_i$.
2. Wähle eine Task T_i , deren Deadline (noch) nicht modifiziert wurde, aber deren unmittelbare Nachfolger alle modifizierte Deadlines besitzen.
Wenn es keine solche Task gibt: EXIT.
3. Setze $d_i^* = \min [d_i, \min(d_N^* - \Delta e_N : T_i \rightarrow T_N)]$.
4. Gehe nach Schritt 2



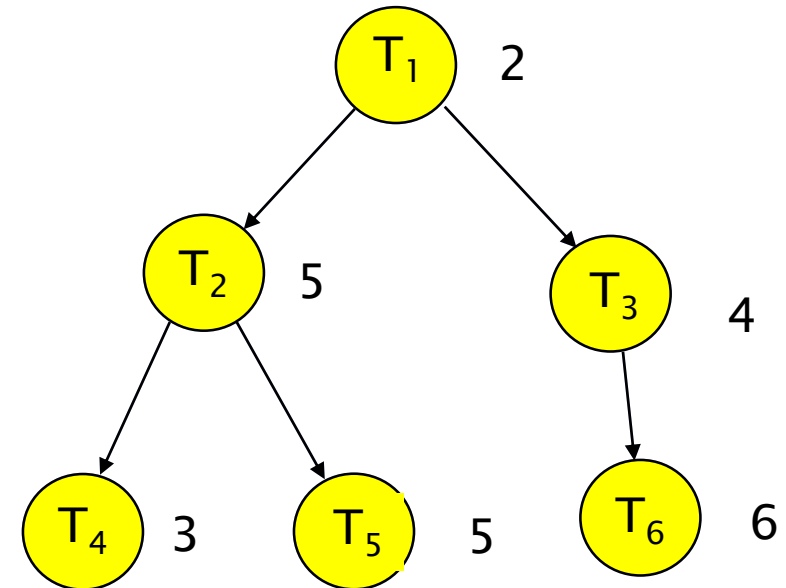
Beispiel

Ursprüngliche Taskparameter

	T_1	T_2	T_3	T_4	T_5	T_6
Δe_i	1	1	1	1	1	1
r_i	0	0	0	0	0	0
d_i	2	5	4	3	5	6

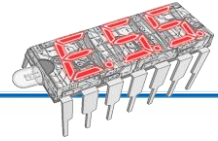
Modifizierte Taskparameter

	T_1	T_2	T_3	T_4	T_5	T_6
Δe_i	1	1	1	1	1	1
r_i	0	1	1	2	2	2
d_i	1	2	4	3	5	6



$$r_i^* = \max [r_i, \max(r_v^* + \Delta e_v : T_v \rightarrow T_i)]$$

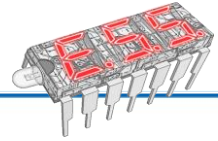
$$d_i^* = \min [d_i, \min(d_N^* - \Delta e_N : T_i \rightarrow T_N)]$$



Gegenüberstellung

- Optimal für Scheduler
- Optimal für Kostenfunktion

	sync. activation	preemptive async. activation	non-preemptive async. activation
independent	EDD (Jackson '55) $O(n \log n)$ Optimal	EDF (Horn '74) $O(n^2)$ Optimal	Tree search (Bratley '71) $O(n n!)$ Optimal
precedence constraints	LDF (Lawler '73) $O(n^2)$ Optimal	EDF * (Chetto et al. '90) $O(n^2)$ Optimal	



Planen periodischer Tasks

Annahmen:

1. Alle Tasks mit harter Deadline sind periodisch.
2. Die Tasks sind unterbrechbar.
3. Die Deadlines entsprechend den Perioden.
4. Alle Tasks sind voneinander unabhängig.
5. Die Zeit für einen Kontextwechsel ist vernachlässigbar.
6. Für einen Prozessor und n Tasks gilt die folgende Gleichung bzgl.

Der durchschnittlichen Auslastung :

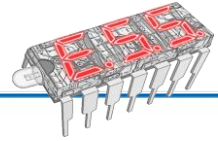
$$U = \sum_{(i=1, \dots, n)} (\Delta e_i / \Delta p_i)$$

Wenn $U = U_{UB}(\{T\}, A)$, dann ist der Prozessor voll ausgelastet.

Darüber hinaus ist kein Plan möglich.

Idee:

Es wird kein expliziter Plan aufgestellt, der (zeitbasiert) auf Fristen oder Spielräumen beruht, sondern es existiert ein impliziter Plan, der durch eine Prioritätszuordnung repräsentiert wird.



Rate Monotonic Scheduling

Definition

- Rate einer periodischen Task
= Anzahl der Perioden im Beobachtungszeitraum
= Frequenz (über unbegrenzte Zeitraum)

- Prioritätsordnung:

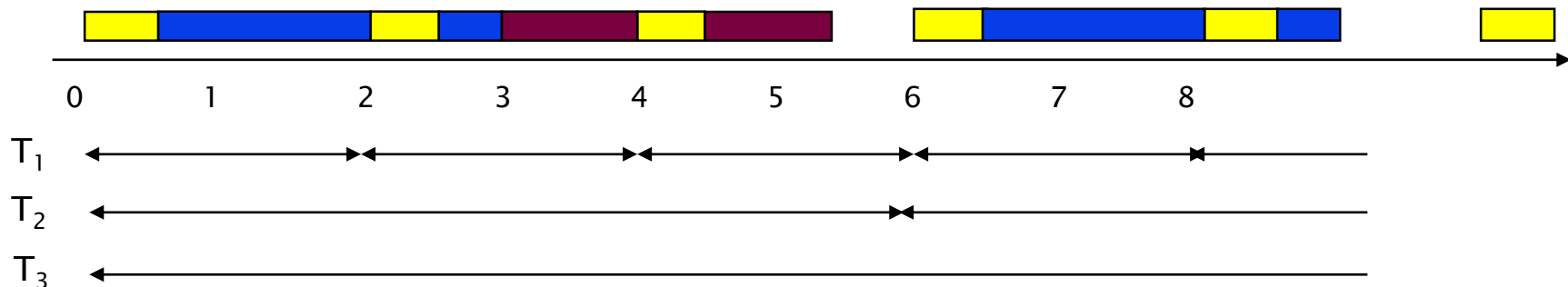
$$rms(i) < rms(j) \text{ für } 1 / \Delta p_i < 1 / \Delta p_j$$

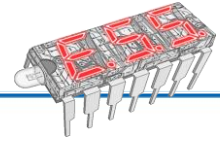
$$\Delta e_1 = 0,5, \Delta p = 2$$

$$\Delta e_2 = 2, \Delta p = 6$$

$$\Delta e_3 = 2, \Delta p = 10$$

- Beispiel





Grenzen des Verfahrens

Ausgangssituation: für diese Werte findet RMS einen Plan. Die Auslastung liegt des Prozessors liegt bei 0,828.

$$T_1 : \Delta e_1 = 3, \Delta p_1 = 7$$

$$T_2 : \Delta e_2 = 2, \Delta p_2 = 5$$

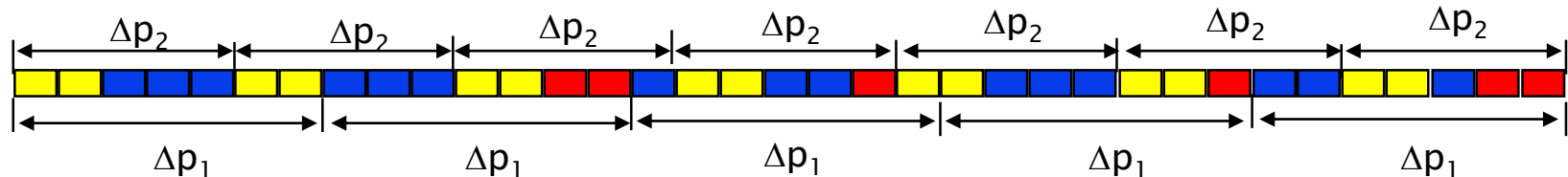


$$kgV = 35$$

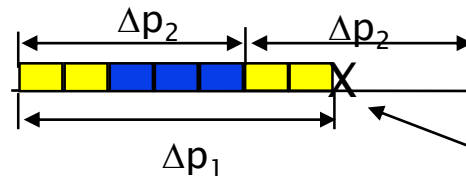
$$T_1 : 15 \text{ Zeiteinheiten}$$

$$T_2 : 14 \text{ Zeiteinheiten}$$

$$29/35 = 0,828$$



Situation 2: Erhöhung des Rechenbedarfs von T_1 um 1 Einheit ($\Delta e_1 = 4$). RMS kann nicht mehr angewandt werden.

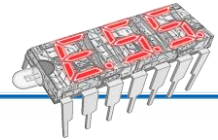


$$T_1 : 20 \text{ Zeiteinheiten}$$

$$T_2 : 14 \text{ Zeiteinheiten}$$

$$34/35 = 0,971$$

T_1 hat seine Deadline verletzt, da es bis zum Ende seine Intervalls keine 4 Zeiteinheiten zur Verfügung hatte.



Obere Schranke für RMS

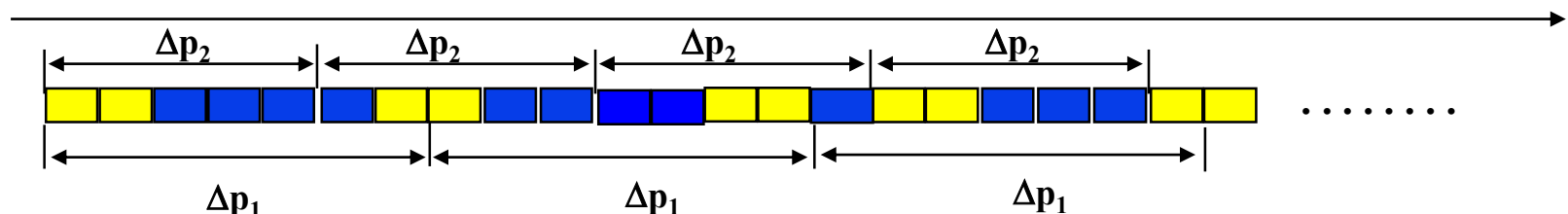
Frage: Gibt es eine obere Schranke U_{lub} der Prozessorauslastung, für die immer ein Plan nach RMS garantiert werden kann (d.h. ein hinreichendes Kriterium für die Einplanbarkeit) ?

U_{lub} ist die Auslastung, für die RMS optimal ist, d.h. einen Plan findet, wenn überhaupt einer existiert. Es kann natürlich Verfahren geben, die eine bessere Auslastung realisieren.

Nach (Liu, Layland, 1973) gilt für n Tasks: $U_{lub} = n (2^{1/n} - 1)$.

Für $n = 1$:	U_{lub}	=	1
Für $n = 2$:	U_{lub}	=	0,828
Für $n \rightarrow \infty$:	$\lim U_{lub}(n) = \ln(2)$	=	0,693

Für EDF ist das kein Problem, da das Verfahren den Prozessor bis 1 auslasten kann.



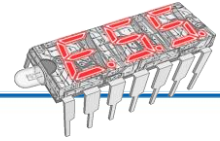
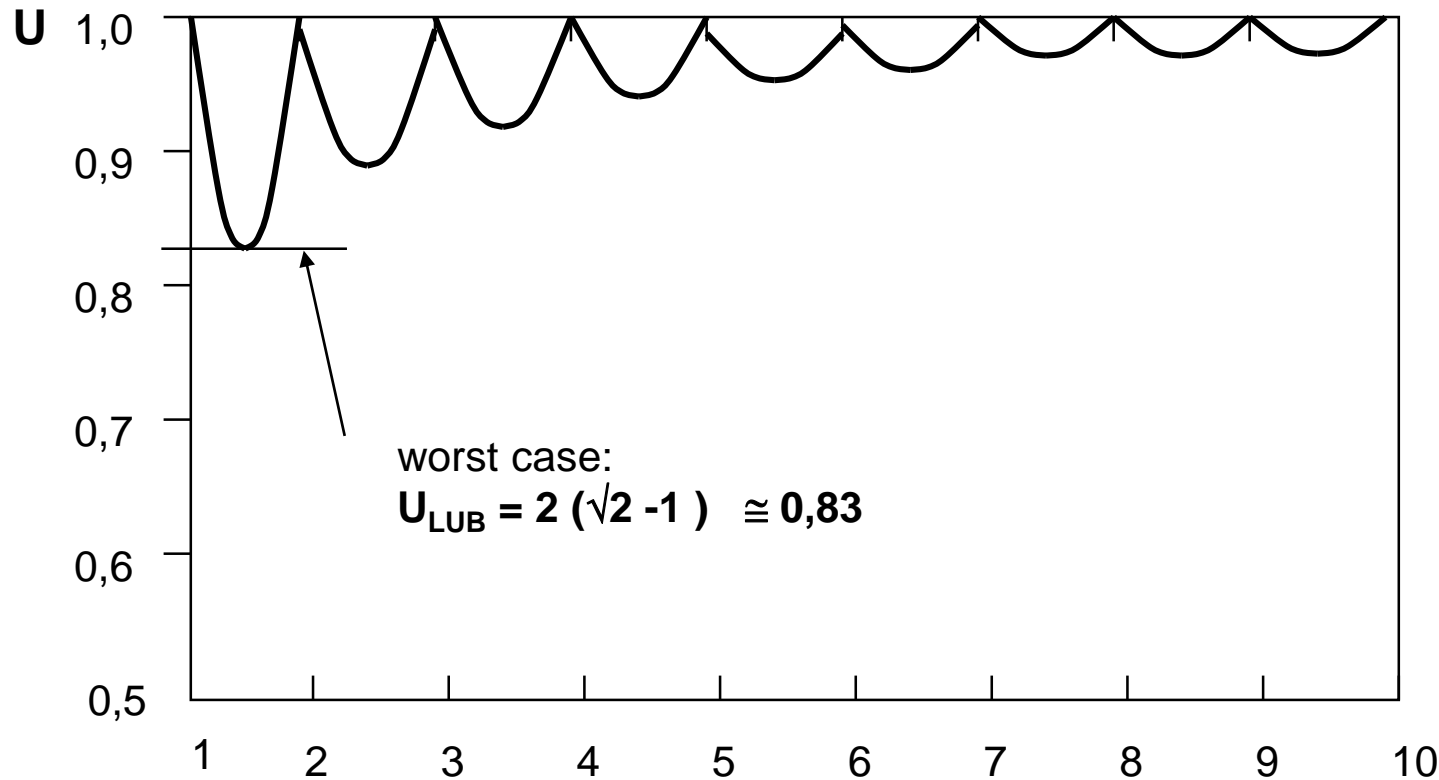
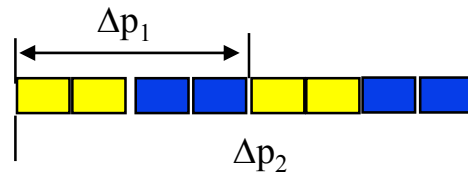


Abbildung auf die Perioden für 2 Tasks

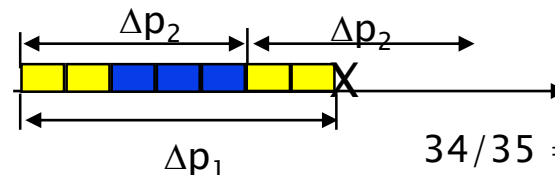


$$k = \Delta p_2 / \Delta p_1$$

harmonische Perioden
 $k \in \mathbb{Z}$



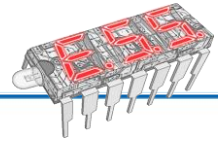
nicht-harmonische Perioden
 $k \in \mathbb{R}$



$$T_1 : \Delta e_1 = 4, \Delta p_1 = 7$$

$$T_2 : \Delta e_2 = 2, \Delta p_2 = 5$$

$$34/35 = 0,971$$



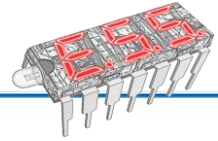
Zusammenfassung RMS

Für alle Ausführungszeiten *und* Periodenverhältnisse von n Tasks wird unter RMS ein gültiger Plan gefunden, wenn die Auslastung die Schranke von $n \cdot (\sqrt[n]{2} - 1)$ nicht übersteigt.

RMS ist einfacher zu realisieren als EDF, die Prioritäten anhand der Perioden werden einmal zu Beginn festlegen.

Aber RMS ist nicht immer in der Lage eine Lösung zu finden, obwohl eine existiert.

harte Echtzeit		weiche Echtzeit	
synchron bereit		asynchron bereit	
periodisch	aperiodisch	sporadisch	
präemptiv		nicht-präemptiv	
unabhängig		abhängig	
statisch		dynamisch	

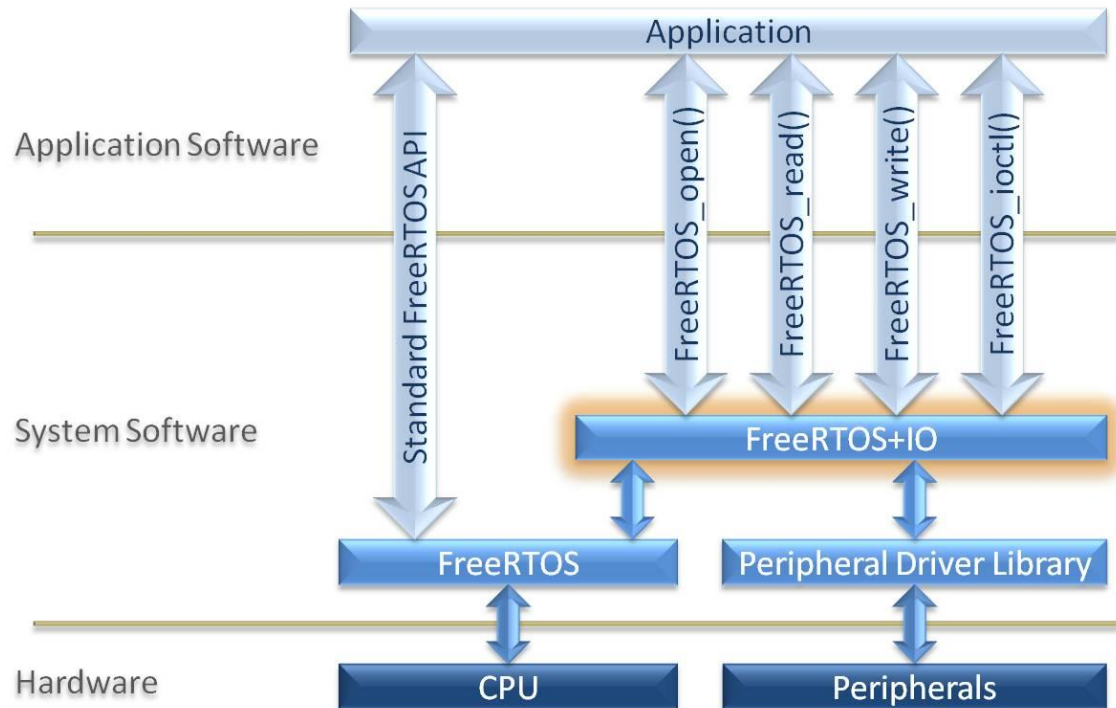


Umsetzung in eingebetteten Geräten

RTOS

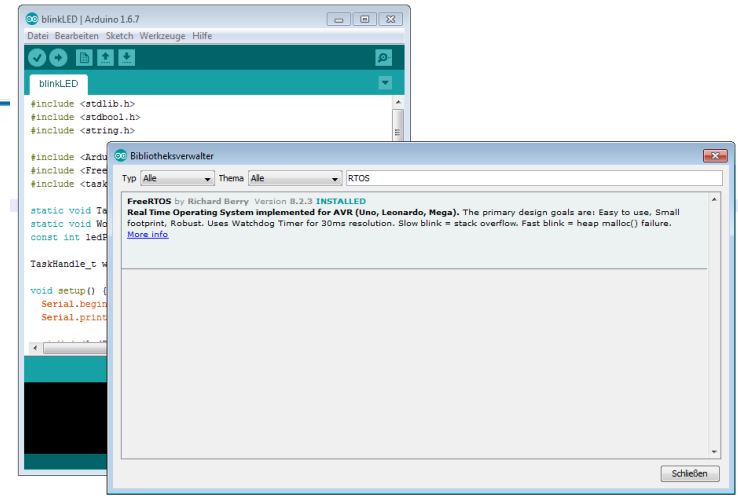
Executive

Scheduler



Auswahlprozess

Arduino
FreeRTOS



Anwend-
barkeit

no scheduling

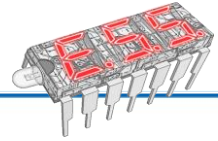
Small size RTOS
FreeRTOS

μLinux

Real Time Linux

Processing Power

Source: Präsentation von Richard Berry (Gründer FreeRTOS) auf Youtube



Vorteile

- RTOS vs. Super Loop Design
 - Trennung von Funktionalität und Timing
 - Definition von Prioritäten
- Vorteile beim Einsatz eines RTOS
 - Deutliche bessere Skalierbarkeit als im SLD
 - RTOS entlastet den Nutzer und behandelte Timing, Signale und Kommunikation
 - Mischung von Hard- und Softrealtime Komponenten
 - Verbesserte Wiederverwendbarkeit des Codes insbesondere bei Hardwarewechseln

ACHTUNG!

Ein RTOS ist kein Garant für ein fehlerfreie Programmabarbeitung!

Vgl. Mars Pathfinder 1997

Grundfunktionalität

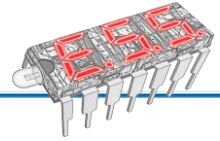
- Task structure
- Task creation
- Task scheduling
- Priority setting

```
static void WorkerTask(void *pvParameters)
{
    static uint32_t idelay;
    static uint32_t Delay;
    Delay = 10000000;
    /* Worker task Loop. */
    for(;;)
    {
        /* Simulate work */
        Serial.println("Worker started");
        for (idelay = 0; idelay < Delay; ++idelay);
        //Serial.println("Worker finished");
        /* Suspend Task */
        //vTaskSuspend(worker_id);
    }
    /* Should never go there */
    vTaskDelete(worker_id);
}
```

Preemptives Scheduling

Kooperatives Scheduling

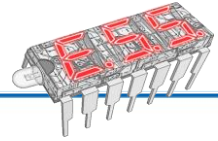
```
static void TaskBlinkRedLED(void *pvParameters) // Main LED Flash
{
    TickType_t xLastWakeTime;
    xLastWakeTime = xTaskGetTickCount();
    while(1)
    {
        //Serial.println("Job1");
        digitalWrite(ledPin, HIGH);
        vTaskDelayUntil( &xLastWakeTime, ( 100 / portTICK_PERIOD_MS ) );
        digitalWrite(ledPin, LOW);
        vTaskDelayUntil( &xLastWakeTime, ( 400 / portTICK_PERIOD_MS ) );
    }
}
```



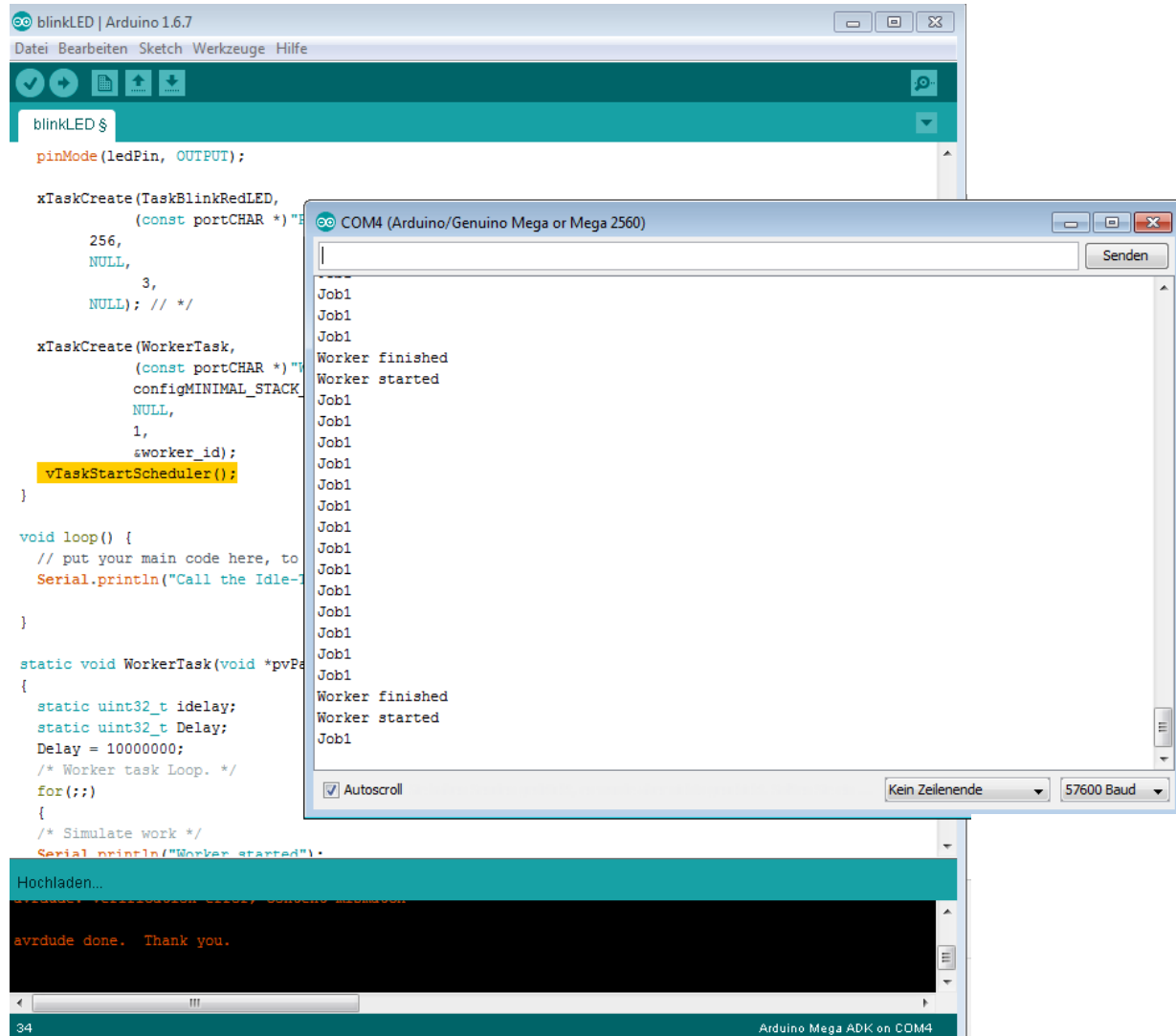
Grundfunktionalität

- Task structure
- Task creation
- Task scheduling
- Priority setting

```
void setup() {  
    Serial.begin(57600);  
    Serial.println("Configuration started");  
  
    pinMode(ledPin, OUTPUT);  
  
    xTaskCreate(TaskBlinkRedLED,  
                (const portCHAR *) "RedLED",  
                256,  
                NULL,  
                3,  
                NULL); // */  
  
    xTaskCreate(WorkerTask, // Function name  
                (const portCHAR *) "Worker", // Task name  
                configMINIMAL_STACK_SIZE+1000, // heap size  
                NULL, // Parameters  
                1, // Priority  
                &worker_id); // Task handle  
    vTaskStartScheduler();  
}
```



Ergebnis



The screenshot displays the Arduino IDE interface. The main window shows the 'blinkLED' sketch in the 'File' menu. The code is as follows:

```
pinMode(ledPin, OUTPUT);

xTaskCreate(TaskBlinkRedLED,
            (const portCHAR *) "
            256,
            NULL,
            3,
            NULL); // */

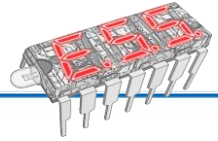
xTaskCreate(WorkerTask,
            (const portCHAR *) "
            configMINIMAL_STACK,
            NULL,
            1,
            &worker_id);
vTaskStartScheduler();
}

void loop() {
    // put your main code here, to
    Serial.println("Call the Idle-");
}

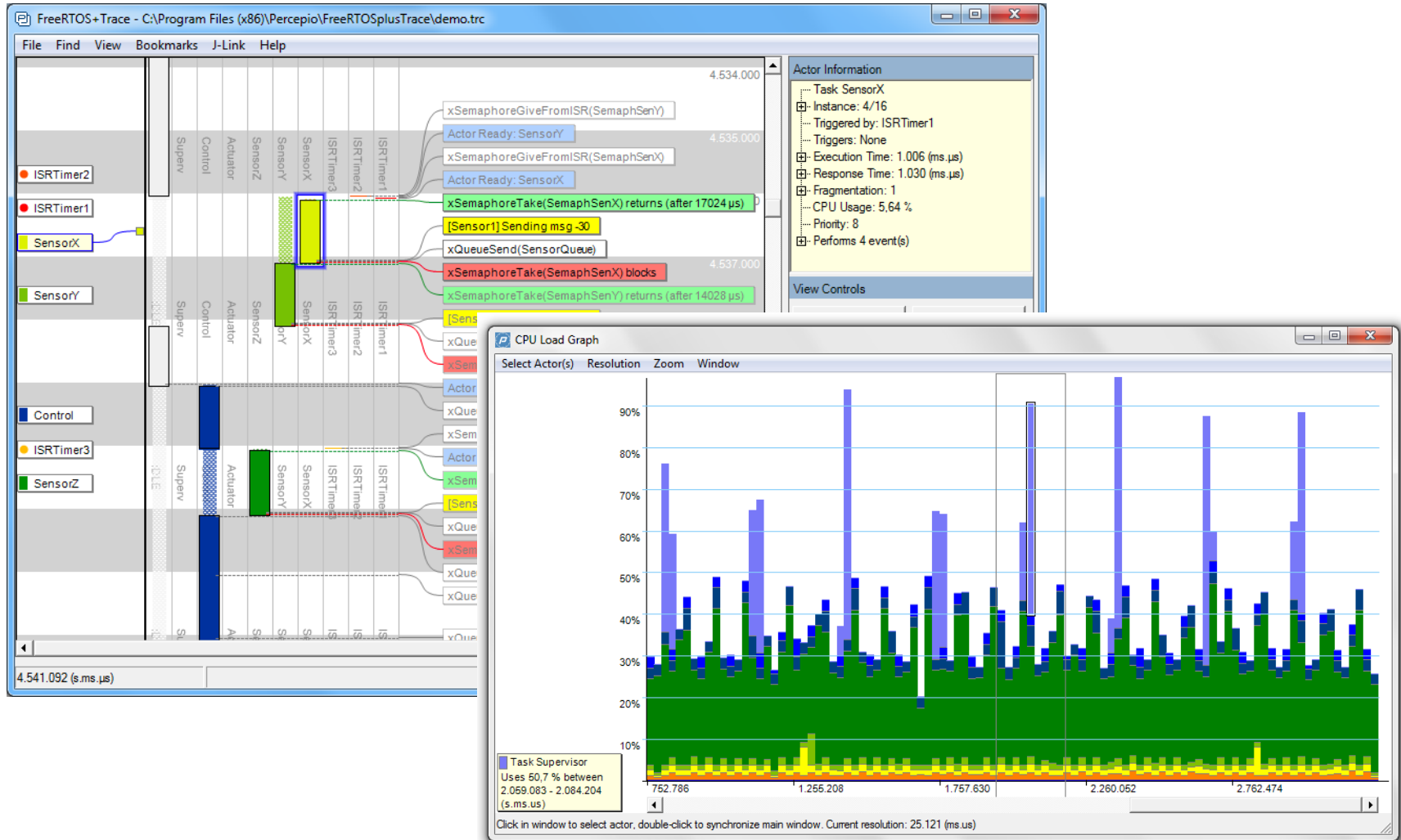
static void WorkerTask(void *pvPa
{
    static uint32_t idelay;
    static uint32_t Delay;
    Delay = 10000000;
    /* Worker task Loop. */
    for(;;)
    {
        /* Simulate work */
        Serial.println("Worker started");
    }
}
```

The serial monitor window, titled 'COM4 (Arduino/Genuino Mega or Mega 2560)', shows the output of the program. It displays a series of 'Job1' messages, followed by 'Worker finished' and 'Worker started' messages, indicating the execution of the tasks. The monitor is set to 'Kein Zeilenende' and '57600 Baud'.

Below the serial monitor, the 'Hochladen...' button is visible, and the status bar at the bottom indicates '34' and 'Arduino Mega ADK on COM4'.



Analyse und Evaluation



Bis zur nächsten Woche ...

www.ovgu.de