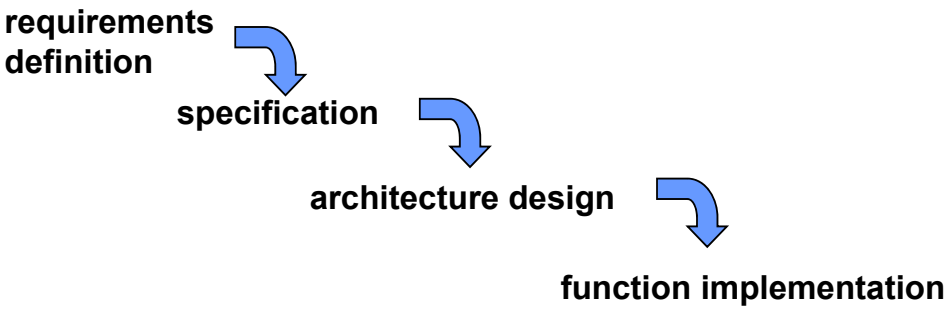
# 5. Systematische Integration

## 5.1 Entwurfsprozess eingebetteter Systeme

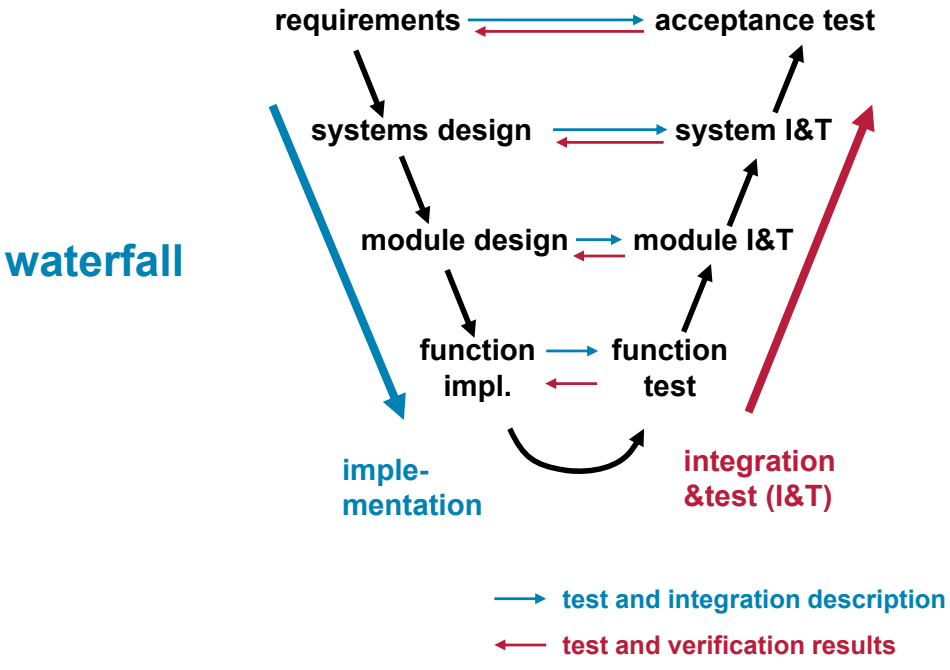
- **der klassische Systementwurf verwendet das Wasserfallmodell (Waterfall Model)**




- **das Wasserfallmodell betrachtet den Entwurf als Sequenz von Implementierungsschritten (Top-Down-Design)**
- **es fehlt ein systematischer Ansatz für Integration und Test**
  - wichtig für komplexe Systeme und für Zulieferketten (Supply Chains)



## Entwurfsprozess – V-Model

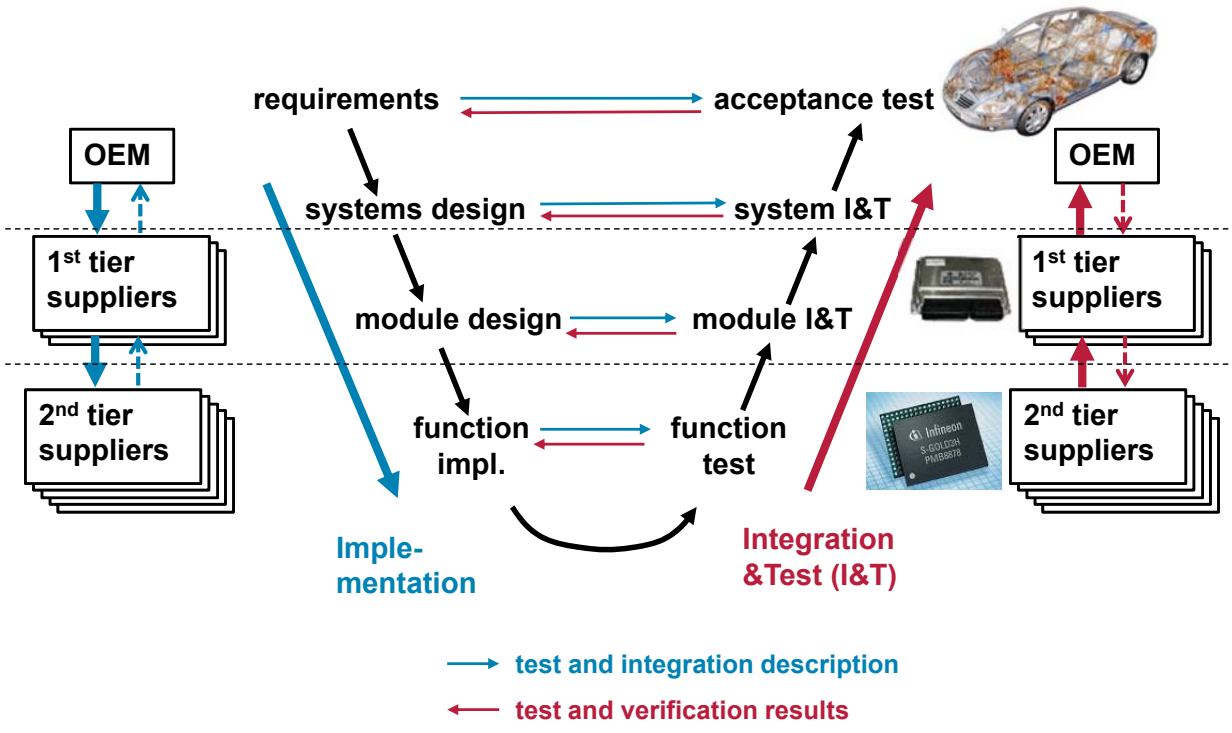


## V-Model

- **das V-Model erweitert das Wasserfallmodell**
  - durch einen umfassenden Test-/Verifikations- und Integrationsprozess
  - spiegelt den Implementierungsprozess des Wasserfallmodells
- **das V-Model erlaubt die systematische Einbettung von Entwurfsprozess und Entwurfsergebnis ("Artefact")**
  - Entwurf von Software und Plattform werden Teil des Entwurfs eines eingebetteten Systems
  - Entwurf eines eingebettete Systems wird Teil des Entwurfs eines Fahrzeugs, eines Flugzeugs, einer Industrieanlage oder eines Gebäudes
- **das V-Modell ist De-Facto-Standard im heutigen Systementwurf**



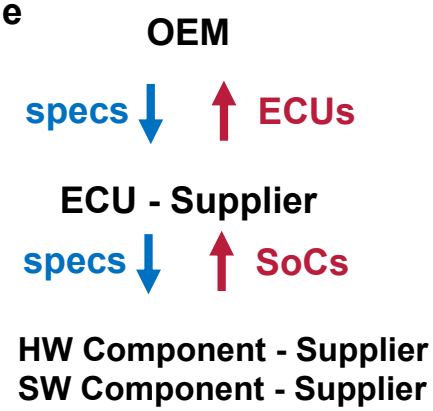
## Einbettung von Entwurfsprozess und Artefact im V-Model



## V Model und Modellgestützter Entwurf

- **zwischen je zwei Ebenen gibt es einen Übergang zwischen Entwurfsteams und Verantwortlichkeiten**
- **auf jeder Ebene werden ausreichende Informationen benötigt, um das Entwurfsergebnis (Artefact) unabhängig überprüfen zu können**
- **favorisiert werden formalisierte Modelle**
  - abstrahieren von unnötigen Details
  - verringern Unklarheiten in der Implementierung
  - erlauben unabhängigen Test, Verifikation und Funktionsvalidierung
  - unterstützen die Nutzung von Verträgen (Contracting)

→ ***modellgestützter Entwurf***






## Anforderungen an eine systematische Integration

- **die systematische Integration basiert auf einer modularen und vorhersagbaren Implementierung und Modellierung der Funktion**
- **angestrebt werden zwei Eigenschaften**
  - **Composability**
    Bei der Integration behält eine Funktion/ein Modul das in Modellen beschriebene Verhalten **unverändert** bei
  - **Compositionality**
    Das aus der Integration resultierende Verhalten ist aus den Modellen der einzelnen Funktionen/Module ohne weitere Kenntnisse **formal ableitbar**
- **beide Eigenschaften ermöglichen eine modellgestützte modulare Integration**
  - für Composability genügt die Kenntnis der abstrakten Funktion und ihrer Schnittstellen, Compositionality erfordert zusätzlich Methoden zur Komposition des Verhaltens

## Voraussetzungen für Composability und Compositionality

- **Voraussetzung sind stets geeignete Modelle *und* geeignete Plattformen**
- **beide Eigenschaften beziehen sich auf bestimmte Entwurfsaspekte und deren Modelle**
- **Entwurfsaspekte bei der Integration eingebetteter Systeme**
  - **Integration der logischen Funktion**
    - Trennung über Isolation und Steuerung der Zugriffe auf Daten und Rechenschritte (Kap. 4)
    - Ziel: **Composability**; Compositionality bei Abhängigkeit der Funktionen
    - *Problem der Funktionsentwicklung - in der VL nicht tiefer betrachtet*
  - **Beherrschung des Zeitverhaltens bei der Integration** (Kap.4):
    - Ziel: **Composability** oder **Compositionality**
    - stark von der Plattform beeinflusst - **Fokus der VL**

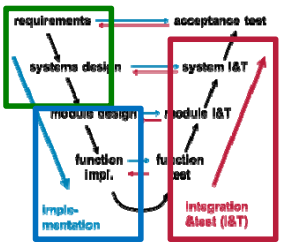


## 5.2. Zeitverhalten – Grundlagen

**was uns interessiert:**

- **Wie kann das Zeitverhalten kontrolliert werden?**
  - **Scheduling**
- **Kann die Plattform die Anwendungen mit dem geforderten Zeitverhalten ausführen?**
  - **Schedulability**
- **Beeinflusst das Zeitverhalten die Funktion?**
  - **Cause-effect Chains** (Wirkketten)
    (in VL nur skizziert)

## Entwurfsprozess - Wo wird das Zeitverhalten benötigt?




- **in den frühen Phasen des Entwurfs**
  - **Ziel:** Auslegung der Vernetzung und des physikalischen Aufbaus
  - **Datenquellen:** Funktionsarchitektur, Schätzung, frühere Implementierung
  - **Problemstellung:** oft noch Änderungen unterworfen
- **im Modulentwurf**
  - **Ziel:** Strukturierung der Plattformkomponenten, Lastverteilung (Multicore),
  - **Datenquellen:** Funktionsarchitektur, frühere Implementierung, neue Implementierung, Messung, Schätzung
  - **Problemstellung:** Daten oft nur teilweise bekannt
- **bei Integration und Test des Gesamtsystems**
  - **Ziel:** Adaption an Plattform und Physik, Überprüfung der Anforderungen und der Implementierung
  - **Datenquellen:** Modelle, Messungen, Systemtests
  - **Problemstellung:** Daten oft nur teilweise bekannt, Aufwand



## Problematik des Zeitverhaltens

- **grundsätzlich ist die Bestimmung sehr aufwendig**
  - da **Ausführungszeiten** und – **häufigkeit** von der Funktion und den Daten bestimmt werden, ist der **Zustandsraum der Ausführung** prinzipiell so groß wie der der Funktion
  - hinzu kommt der **Zustandsraum der Plattform** – Architektur, Taktung, …
- **Vorhersagbarkeit der Plattform wird mit komplizierten Architekturen schwieriger**
  - Architekturen i.d.R. **nicht auf Vorhersagbarkeit optimiert**, sondern auf Performanz
- **wieviel muss man wissen, um die interessanten Fragen zu beantworten**
  - wieviel **Vorhersagbarkeit** benötigen wir?
  - Was ist eine **zulässige *und* sinnvolle Abstraktion**?

## Bestimmung des Zeitverhaltens

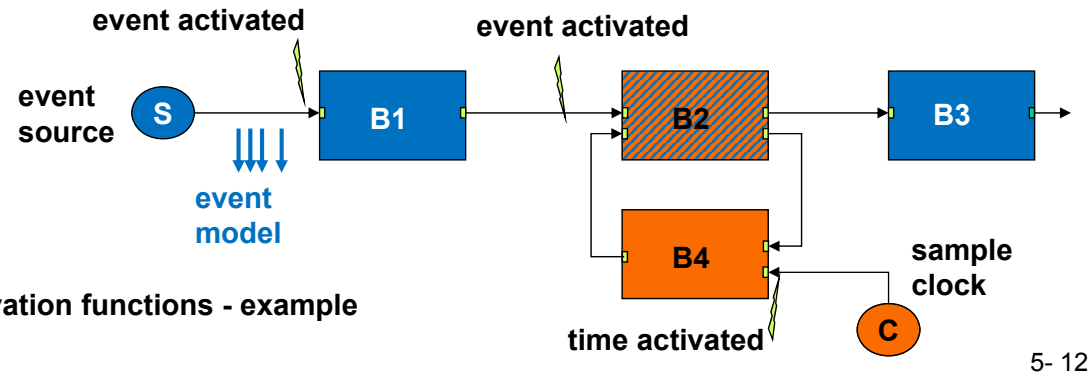
- **prinzipiell mit allen bekannten Verifikationsverfahren Test, Simulation, Prototypen, formale Analysen**
  - grundsätzlich sehr aufwendig und **generell riskant** wg. Zustandsraum und fehlenden Daten
  - Modelle in frühen Phasen **oft nicht ausführbar** – keine Simulation oder Test möglich
  - **Formale Analysen** problematisch wg. **Komplexität** und **Vollständigkeit** der Modelle und Methoden

→ ***benötigen sinnvolle und zulässige Abstraktion***



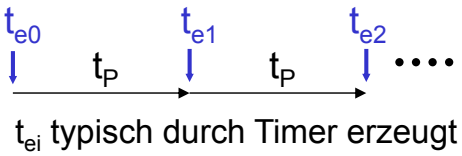
## 5.2 Ereignismodelle und Last

- **die Aktivierung von Tasks erfolgt zeitgesteuert oder ereignisgesteuert**
- **in beiden Fälle kann die Aktivierung von Tasks durch eine Sequenz einzelner Ereignisse in der Zeit modelliert werden**
  - auch Basis der Simulation
- **für formale Betrachtungen ist es günstiger, von einzelnen Ereignissen zu abstrahieren und die Eigenschaften der (unendlichen) Ereignissequenz zu betrachten**
  - eine solche Sequenz wird als **Ereignisstrom (event stream)** bezeichnet
  - er wird mit abstrakten **Ereignis(strom-)modellen** beschrieben




activation functions - example

## Abstrakte Ereignismodelle - periodische Ereignisse




tei typisch durch Timer erzeugt

$$t_{e_{i+1}} - t_{e_i} = t_P$$

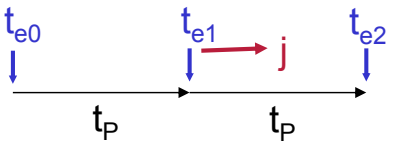
$$t_{e_i} = t_{e_0} + i \cdot t_P$$

- **dies ist das wichtigste Ereignismodell, das bei allen periodischen Aktivierungen zeitaktivierter Tasks auftritt (Kap. 3)**
- **die Sequenz ist prinzipiell unendlich**
- **Bespiele**
  - **periodische Abtastung**: SDF, Simulink, …
  - **periodisches Polling** (Kap. 3)



## Periodische Ereignisse mit Jitter

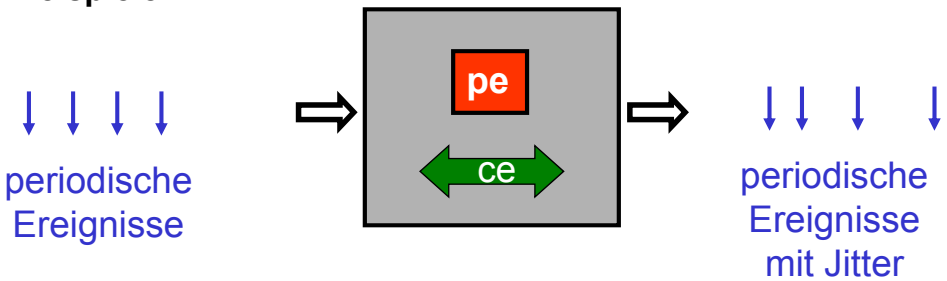
- **der Jitter j verschiebt den Ereigniszeitpunkt um maximal eine Periode**




$$t_{e_0} + i \cdot t_P < t_{e_i} < t_{e_0} + i \cdot t_P + J$$

$$J < t_P$$

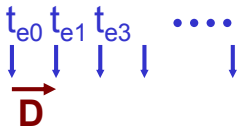
- **Beispiele**




- Ursachen
  - datenabhängige Ausführungszeiten
  - Schedulingeffekte

## Periodische Ereignisse mit Jitter und Burst

- **der Jitter kann zu zwei beliebig dicht aufeinanderfolgenden Ereignissen führen**
- **wird der Jitter größer als die Periode, J > $t_P$ können sich Bündel von Ereignissen mit beliebiger Dichte bilden ("Burst")**
- **die tatsächliche Last wird dann durch die Plattformeigneschaft begrenzt, etwa die Zahl der maximal übertragbaren Telegramme oder der maximal ausführbaren Tasks**
- **diese Begrenzung kann durch die maximale Ereignisdichte, D, erfasst werden**




$$t_{e_0} + i \cdot t_P < t_{e_i} < t_{e_0} + i \cdot t_P + J$$

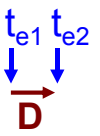
$$t_{e_{i+1}} - t_{e_i} \geq D$$



## Nicht-periodische Ereignismodelle

- **nicht-periodische Ereignisse lassen sich weniger gut beschreiben, da sie oft keinem regelmäßigen Muster folgen**
- **um die auftretende Last beschreiben zu können, führt man auch hier einen minimalen Ereignisabstand D ein**

  **→ sporadisches Ereignismodell**




$$t_{e_{i+1}} - t_{e_i} \geq D$$

- **Beispiele**
  - Eintreffen von Paketen
  - Eingaben einer Benutzerschnittstelle
  - Fehlermeldungen
  - Kommunikation reaktiver Systeme
  - ...

**reaktive Systeme**

## Zusammenfassung Standard-Ereignismodelle

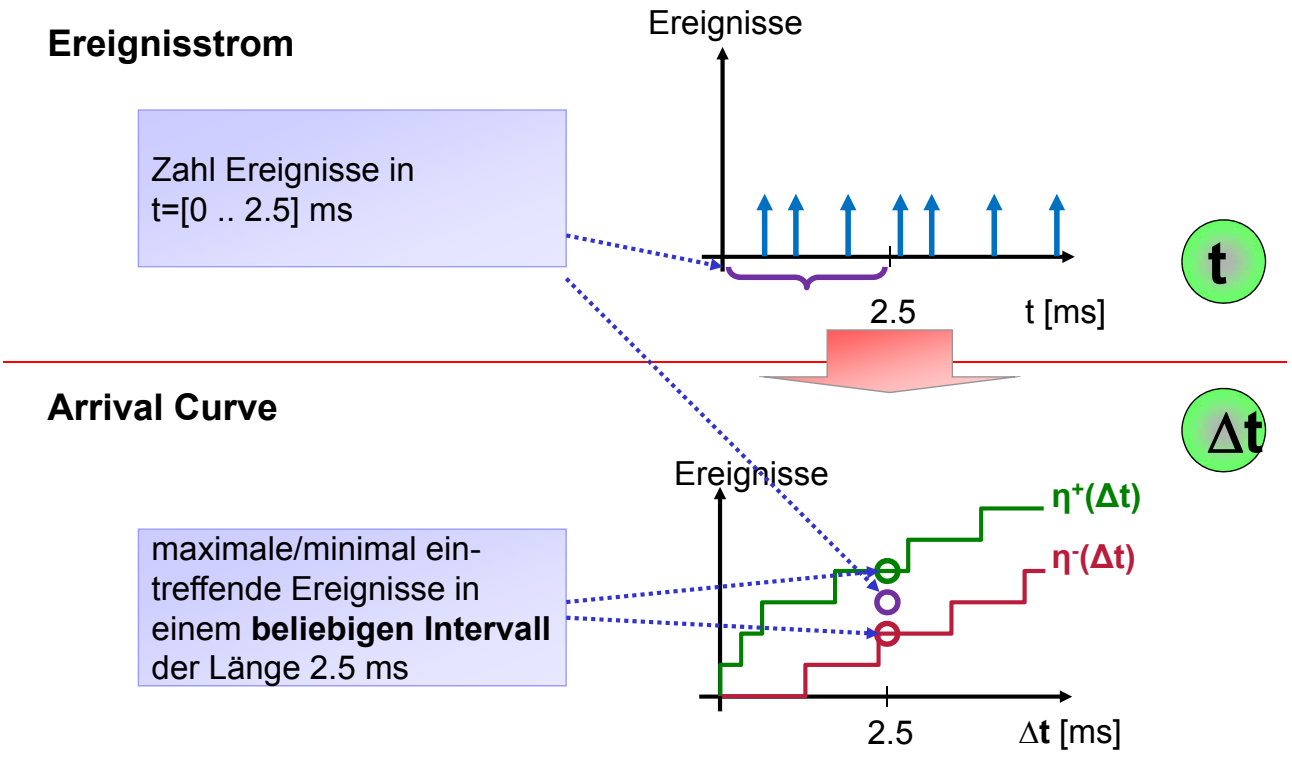
- **die klassischen Standard-Ereignismodelle werden zu einem allgemeinen (P,J,D) – Modell zusammengefasst**
  - extrem kompakte Abstraktion einer unendlichen Sequenz
- **sehr verbreitet genutzt im Gebiet der Echtzeitsysteme**
  - vereinfacht die Analysen
  - erlaubt Garantien (später)
  - sinnvolle und zulässige Abstraktion
- **aber nur beschränkt aussagekräftig**
  - vor allem bei sporadischen Anteilen
  - oder Kombination mehrerer periodischer Ereignisströme (Kommunikation)
  - dann wird das Modell zu konservativ für einen effizienten Entwurf
- **daher wurde in der Kommunikationstechnik ein mächtigeres Modell eingeführt**
  - genauere Beschreibung mit **Arrival Curves**
  - heute auch für eingebettete Systeme verwendet



## Arrival Curves des Network Calculus

- **der Network Calculus transformiert eine Ereignissequenz aus dem Raum der Zeit t in einen Raum über Zeitintervalle $\Delta t$**
  - damit wird von zeitlichen Abhängigkeiten zwischen verschiedenen Ereignissströmen abstrahiert
- **konkret werden zwei Funktionen gebildet**
  - **$\eta^+(\Delta t)$** maximale Zahl aktivierender Ereignisse, die in einem Zeitfenster der Länge $\Delta t$ auftreten können
  - **$\eta^-(\Delta t)$** minimale Zahl aktivierender Ereignisse, die in einem Zeitfenster der Länge $\Delta t$ auftreten können
- **beide Funktionen erreichen unendliche Werte für $\Delta t \rightarrow \infty$**
  - zur Nutzung durch periodische Fortsetzung approximiert
  - deckt das (P, J, D)-Modell ab
- **der Network Calculus liefert eine abstrakte Algebra, mit der das Zeitverhalten für viele Fälle im $\Delta t$ – Raum approximiert werden kann**
  - verwenden nur einige der Eigenschaften, da wir Verfahren in t und in $\Delta t$ kombinieren

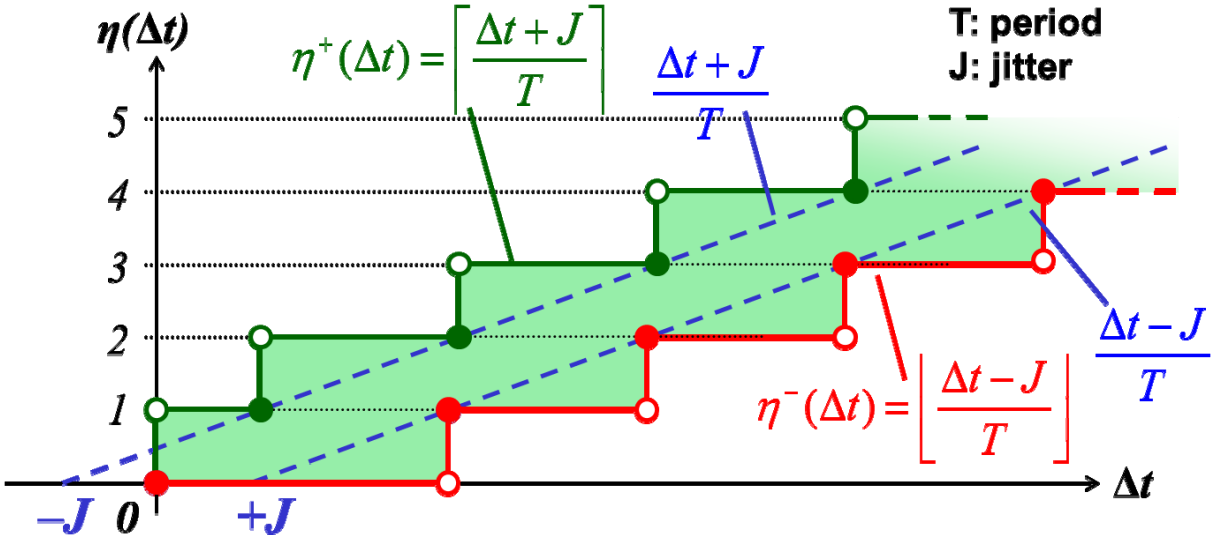
## Ableitung Arrival Curves - Beispiel




## Praktische Eigenschaft der Arrival Curves

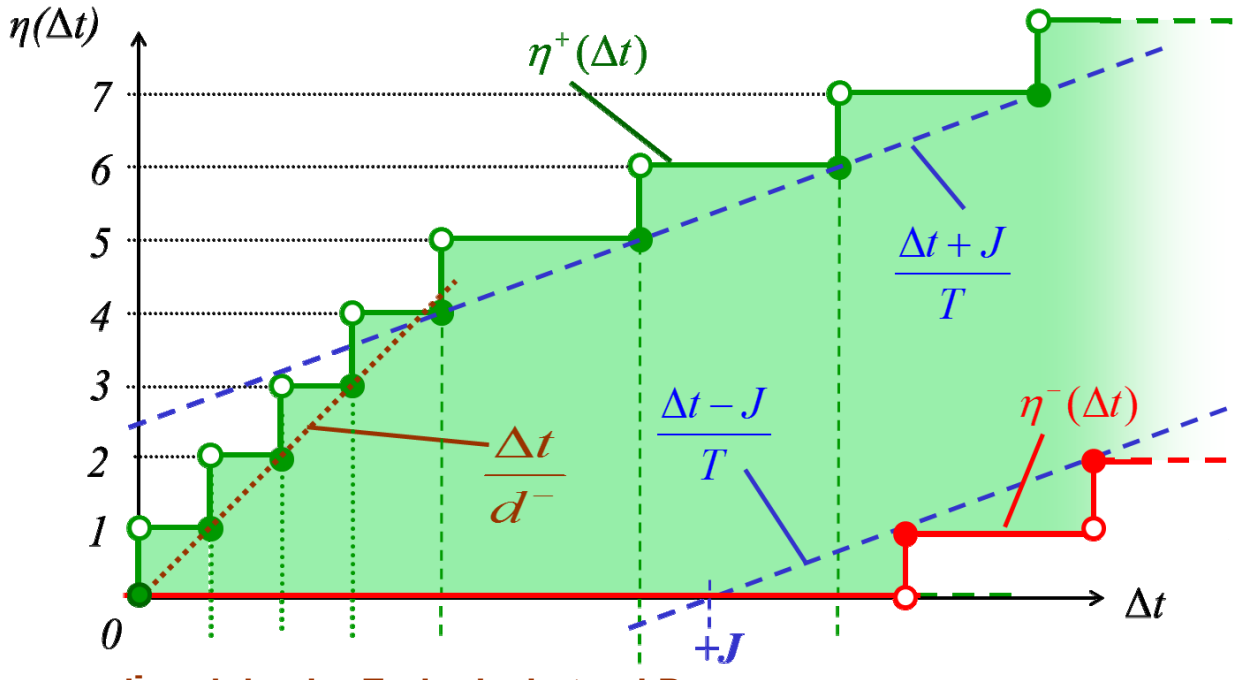
- **Maximale Ereigniskurven sind subadditiv**
  **$\eta^+(\Delta t1 + \Delta t2) \leq \eta^+(\Delta t1) + \eta^+(\Delta t2)$**
  - eine periodische Wiederholung eines Stücks von $\eta^+$ dominiert damit die reale unendliche Kurve
- **Minimale Ereigniskurven sind superadditiv**
  **$\eta^-(\Delta t1 + \Delta t2) \geq \eta^-(\Delta t1) + \eta^-(\Delta t2)$**
  - die reale unendliche Kurve dominiert damit eine periodische Wiederholung eines Stücks von $\eta^-$

## Beispiel: periodische Ereignisse mit Jitter





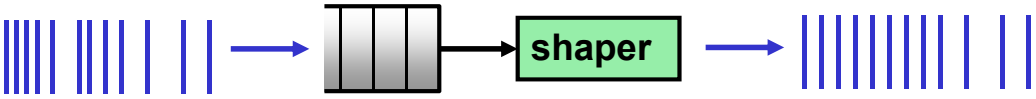

## Beispiel: periodische Ereignisse mit Burst




d⁻ : minimaler Ereignisabstand D

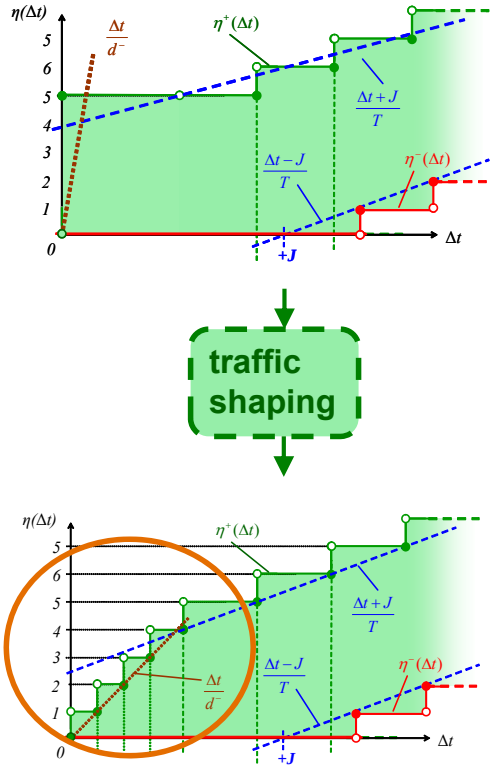
## Beispiel: Traffic Shaping, z.B. AFDX (BAG)

- **Ziel: Begrenzung der Spitzenlast durch Jitter oder Burst**
- **Ansatz: Erzwingung eines minimalen Ereignisabstands D durch „traffic shaping“ – siehe BAG, ähnlicher Ansatz in TSN**
- **verwendet, um Überlast zu vermeiden**
- **erfordert Pufferung und erhöht ggf. die Latenz**



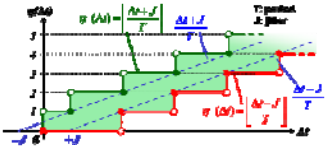


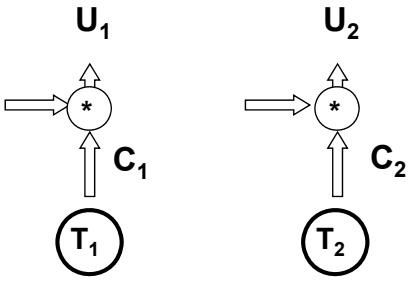

## Traffic shaping - Modellierung und Effekt



## Resultierende Last - Utilization

- **die aktivierenden Ereignisse bestimmen nur die Häufigkeit der Ausführung einer Task in der Zeit t**
- **zur Bestimmung der resultierenden Last wird noch die Ressourcennutzung pro Ausführung benötigt**
- **angegeben wird die Ressourcennutzung C durch**
  - die **Ausführungszeit** (execution time) einer Task oder
  - die **Übertragungszeit** eines Frames
- die **Utilization** ergibt sich aus der Verknüpfung von **C** und **$\eta^+(\Delta t)$**






$U_i$ **Utilization** der Task i

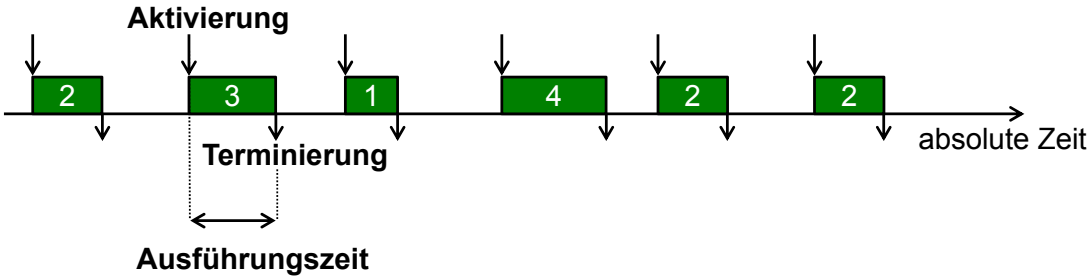
$C_i$ **Execution Time** der Task i



## Utilization

- **in formalen Analyseverfahren wird oft nur die größte Ausführungszeit verwendet – Worst Case Execution Time (WCET)**
  - **vereinfacht das Modell**
- **die Ausführungszeit ist aber nicht konstant und hängt i.d.R. vom Zustand ab (vgl. auch Containertasks in Kap. 3)**
  - **daraus ergibt sich ein Worst Case Execution Trace**

**Ausführungssequenz (Trace) - Beispiel**

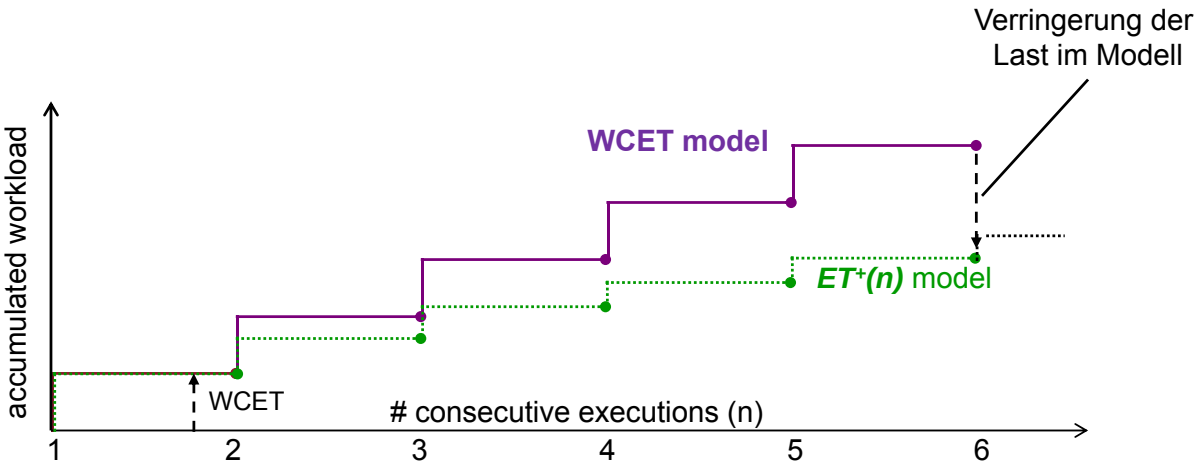


## Workload Model

- **definiere Workload Curve, ebenfalls als Fensterfunktion, dieses Mal über die Ereignisfolge**
  - daraus ergibt sich ein Worst Case Execution Trace
- ***Definition $ET^{-/+}(n)$:*** minimale/maximale akkumulierte Ressourcennutzung über ein Fenster von n aufeinanderfolgenden Ausführungen
  - auch hier gelten Sub- bzw. Superadditivität



## Workload Model: Beispiel



# 5.3 Scheduling - Strategien und Analyse

## Wichtige Begriffe – Zusammenfassung

- **Task: Ausführbare Einheit (vgl. Kap. 3), z.B.**
  - ein Programm, das bis zu seiner Beendigung auf einem Core läuft
  - eine Nachricht, die über ein Kommunikationsnetz gesendet wird
- **Job: Instanz einer Task, z.B.**
  - eine einzelne Ausführung einer wiederholt ausgeführten Task
  - ein einzelnes Frame einer wiederholt gesendeten Nachricht
- **preemptive (unterbrechend): ein Job kann während seiner Ausführung unterbrochen und später fortgesetzt werden**
- **non preemptive (nicht unterbrechend): ein Job kann nicht unterbrochen werden, sobald er gestartet wurde**
- **Worst Case Execution Time (WCET): max. Ausführungszeit eines Jobs**
- **Response Time (Antwortzeit): absolute Zeit von der Aktivierung bis zur vollständigen Ausführung eines Jobs**
- **Worst Case Response Time (WCRT): maximale absolute Zeit von der Aktivierung bis zur vollständigen Ausführung eines Jobs**



## Schedulability

### Begriffe

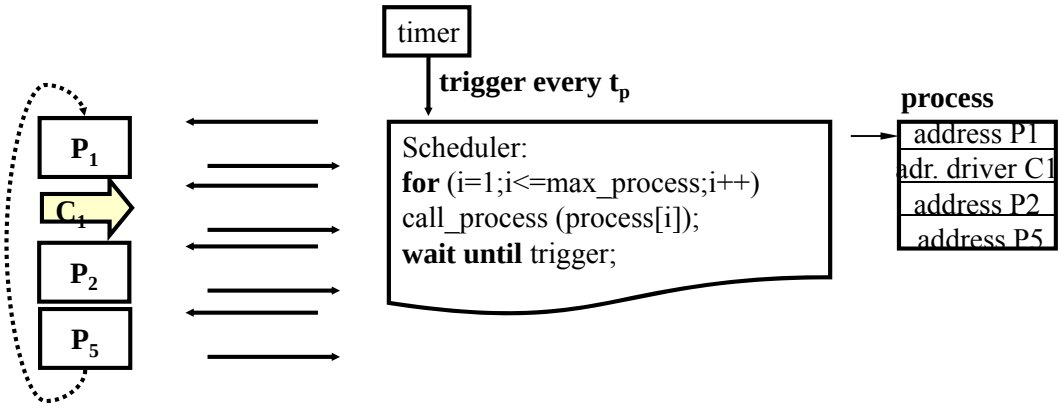
- **Deadline: maximal zulässige Zeit bis zur vollständigen Ausführung eines Tasks**
  - optional: Tasks können Deadlines haben, müssen aber nicht
- **verschiedene Typen von Deadlines**
  - hard deadline: jede Deadline muss eingehalten werden → hard real-time task
  - firm deadline: unter bestimmten Regeln sind gelegentliche Deadline-Misses erlaubt → weakly hard real-time task
  - soft deadline: die Einhaltung von Deadlines bestimmt die Qualität einer Implementierung → soft real-time task
- **Schedulability: ein System ist schedulable (schedulebar), wenn gilt:**
  - alle Jobs aller Tasks halten ihre Deadlines gemäß dem Deadline-Typ ein
  - alle Jobs werden irgendwann einmal ausgeführt und beendet

## Scheduling - Strategien

- **Scheduling ist uns in der VL oft begegnet**
  - hier befassen wir uns systematisch damit
- **Scheduling in Space**
  - exklusive Zuweisung getrennter Ressourcen: Speicherbereiche, Cores, Busleitungen, Frequenzen, …
- **Scheduling in Time**
  - Zuweisung von Ressourcen zu unterschiedlichen Zeiten
    - **static order scheduling** – zyklische Wiederholung einer festen Reihenfolge von Tasks
    - **zeitgesteuertes Scheduling** – zyklische Zuweisung von Zeitschlitzen
    - **prioritätsgesteuertes Scheduling** – Zuweisung gemäß Prioritäten
    - andere, z.B. budgetbasiertes Scheduling – hier nicht näher betrachtet
  - *nur wenige dieser Schedulingstrategien sind in der Praxis verbreitet*
    - *betrachten nur eine Auswahl, um die Prinzipien zu verstehen*

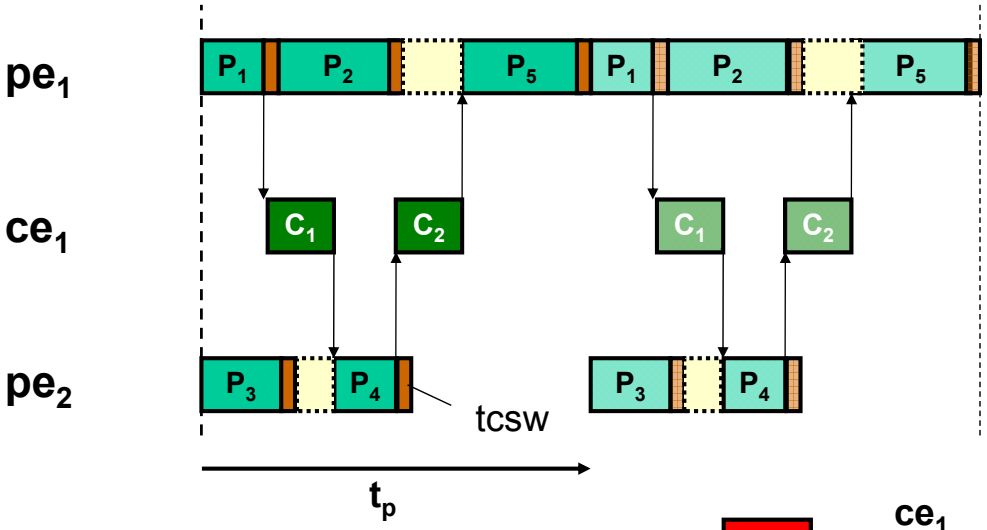


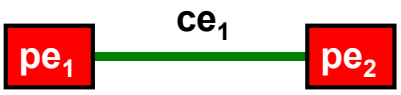
## Static Order Scheduling




- **Durchlaufen einer festen Reihenfolge von Tasks und Kommunikationsschritten**
  - gesteuert durch eine periodisch sequentiell durchlaufene **Prozesstabelle** (vg. Kap. 4: MicroC/OS Background Tasks)
- **Prozesstabelle und Tasks können in einen einzigen Task überführt werden („inlining“), um dann mit dem Compiler optimiert zu werden**

## Static Order - Verzahnung von Abläufen in Komponenten




**architecture example**

**$t_p$: scheduling period**

pe: processing element
ce: communication element

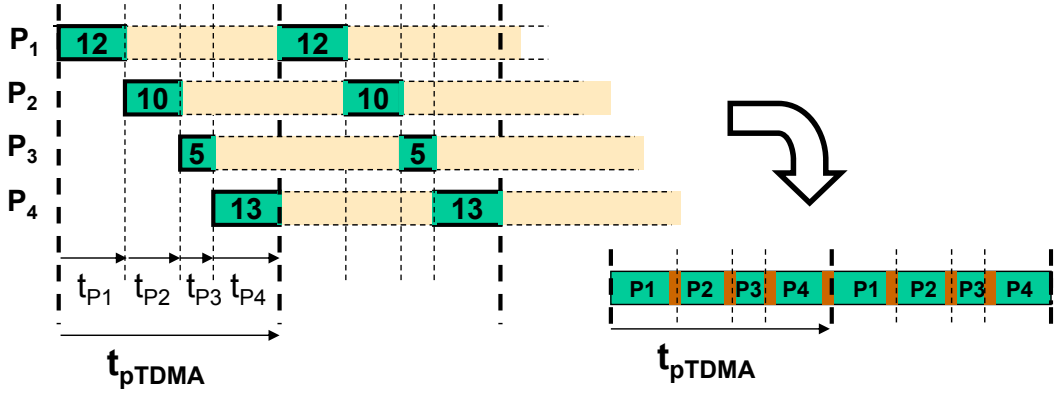


## Static Order Scheduling - Einsatz

- **die exakte Sequenz unterstützt**
  - verzahnte Ausführung (s. Beispiel)
  - Pufferoptimierung (s. Kap. 2)
- **am besten geeignet, wenn Zeitverhalten und Steuerung von Eingabedaten unabhängig sind**
- **wichtige Anwendungen**
  - digitale Signalverarbeitung (DSP)
    z.B. Codegenerierung aus SDFs (Kap 2)
  - einfache Betriebssysteme (s. Kap. 4)
  - Scheduling von Runnables in Containertasks (Kap. 3)
- **Grenzen des Static Order Scheduling**
  - dynamische Umgebungen: Eingaben mit Jitter
  - Ausführung mit datenabhängigem Zeitverhalten

## Zeitgesteuertes Scheduling – statische Slotzuweisung

- **Time Division Multiple Access (TDMA)**
  - periodische Zuweisung fester Zeitslots
  - anwendbar auf pe und ce




**TDMA Beispiel**



## TDMA Time-Partitioning

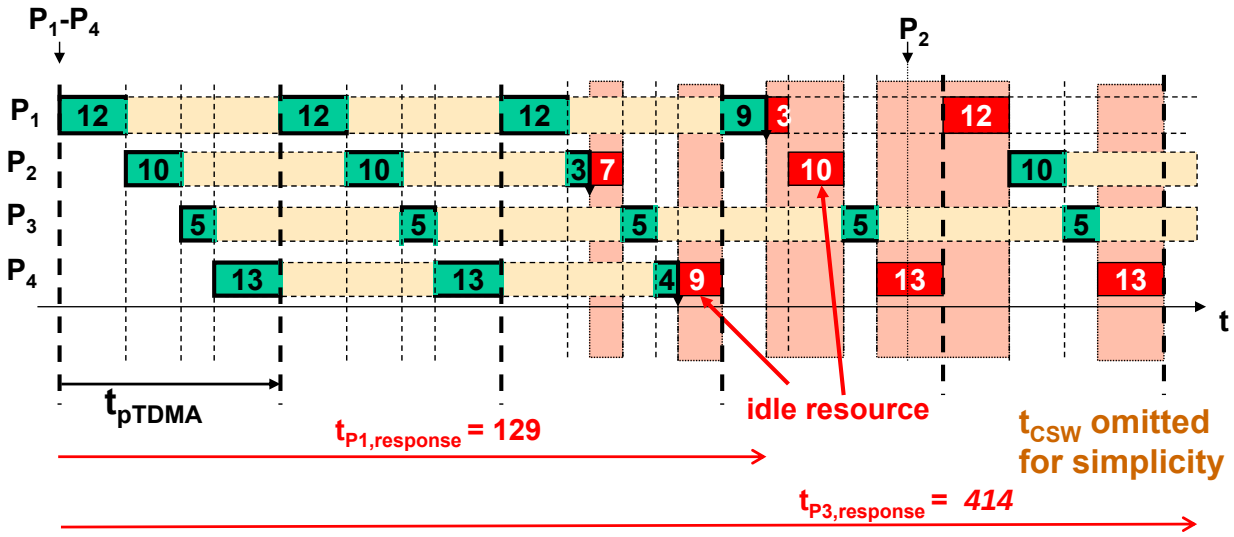
- **Time-Division Multiple Access (TDMA) Scheduling entspricht einer festen Ressource-Partitionierung (Kanaltrennung)**
  - Antwortzeiten werden nur vom Schedule beeinflusst, aber von anderen Funktionen

$$R_i = C_i + (t_{TDMA} - t_{Mi}) \times \left\lceil \frac{C_i}{t_{Mi}} \right\rceil$$

$R_i$ response time frame i; $t_{Mi}$ time slot
$C_i$ execution/frame transmission time
$t_{TDMA}$ TDMA cycle time

- **Annahmen hierbei**
  - alle Sender halten sich an den TDMA Schedule
  - alle Sender sind zeitsynchronisiert
  - das Zeitverhalten $C_i$ wird durch den Zustand der Plattform nicht verändert (z.B. durch Cache-Zustand, TLB, …)
- **TDMA hat begrenzte Effizienz: ungenutzte Slots können nicht von anderen Funktionen genutzt werden**
  - ungenutzte Ressource trotz aktivierten Tasks → **nicht lasterhaltend**

## TDMA Beispiel




**Scheduling and idle times in TDMA**

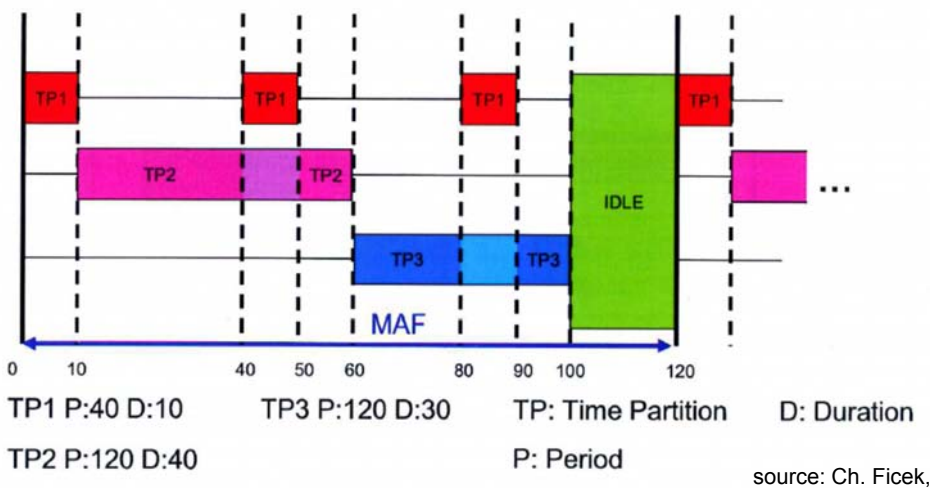


## TDMA in der Praxis

- **robuste Technik**
  - solange die Uhren synchronisiert sind, kann die Steuerung beliebig verteilt werden
  - keine Arbitrierung erforderlich
- **wichtige praktische Beispiele**
  - GSM Protokoll (zellulare drahtlose Netze)
  - FlexRay
  - Profinet
  - ARINC 653 Betriebssystem

## Example: ARINC 653

- **Avionics - IMA**
- **partitions are assigned to time windows TPi iterating over a major Time Window MAF**
- **execution can exceed single time window**
- **supports scheduling hierarchies**



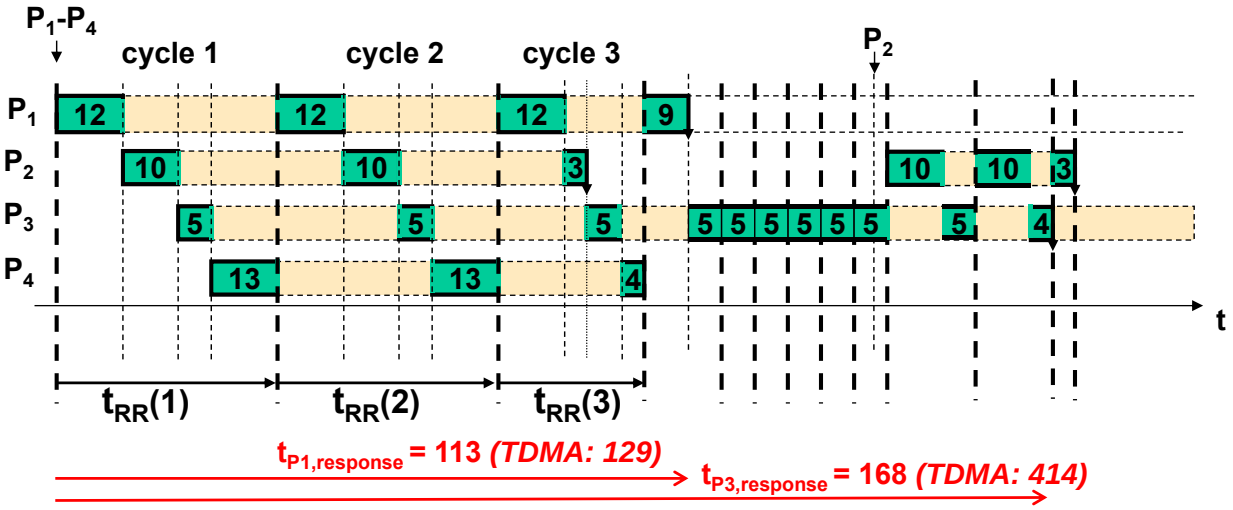

source: Ch. Ficek, Symtavision

## Round-Robin Scheduling

- **Vergabe ungenutzter Zeitslots**
  - keine Leerzyklen – höhere Effizienz als TDMA
- **im ungünstigsten Fall Antwortzeiten wie TDMA**
  - geeignet z.B. für Soft Deadlines oder Funktionen mit möglichst guter QoS (“best effort”)
  - geeignet für Kombinationen von hard real-time und anderen Tasks
- **Unabhängigkeit der Prozesse entfällt**
  - Zeitverhalten komplizierter als TDMA
    - bei Vorliegen der Zeitdaten aber berechenbar
  - aber garantierter minimaler Service

## Round-Robin Beispiel

- **periodische Aktivierung**
- **ungenutzte Ressource unverzüglich freigegeben**
  - keine ungenutzte Ressource bei aktivierten Tasks → **lasterhaltend**




**Round Robin example**



## Round-Robin in der Praxis

- **Round-Robin Steuerung**
  - erfordert Taktsynchronisation wie TDMA
  - zentrale Steuerung wenig aufwendiger als TDMA
  - verteilte Steuerung (Kommunikation) wesentlich aufwendiger
    - welcher Sender ist bereit?
    - wer ist der nächste Sender?
    - Beispiel für Protokoll: Byteflight Minislots (FlexRay dynamisches Segment), dort mit priorisierter Slotvergabe (kann durch RR ersetzt werden)
- **wichtiges praktisches Beispiel**
  - die meisten Betriebssysteme

# Prioritätsgesteuertes Scheduling

- **statisch zugewiesene Prioritäten**
  - die Priorität von Tasks oder Nachrichten bleibt konstant
  - beliebteste Strategie in eingebetteten Systemen (neben TDMA)
  - **static priority preemptive (SPP)**
    - Beispiele OSEK/VDX, AUTOSAR
  - **static priority non preemptive (SPNP)**
    - Beispiele CAN, OSEK/VDX, AUTOSAR
- **dynamisch zugewiesene Prioritäten**
  - Task oder sogar Job Prioritäten zeitlich variabel



# Scheduling mit statischen Prioritäten

- **wir werden uns im Folgenden auf periodische Aktivierung beschränken**
  - Modell 1
    - Deadlines am Ende der Periode
  - Modell 2
    - Deadlines vor dem Ende der Periode
  - Modell 3
    - beliebige Deadlines

## Rate Monotonic Scheduling (RMS) – Modell 1

- **zugrunde liegendes Taskmodell**
  - **periodische Aktivierung** unabhängiger Tasks
  - **Deadline** am Ende der Taskperiode
  - WCET jeder Task ist bekannt
- **Prioritäten werden gemäß Periode zugewiesen**
  - **kürzere Periode → höhere Priorität** - „**Rate Monotonic**“
  - beweisbar optimaler Schedule für Singlecore-Prozessoren
    - optimal: wenn nicht mit RMS schedulebar, dann gar nicht schedulebar unter SPP
  - *Priorität ist ein Parameter des Scheduling und bedeutet nicht Wichtigkeit!*
- **wird breit verwendet, da einfach und leicht analysierbar**
  - eingeführt von Liu und Layland 1973

## Rate Monotonic Scheduling (RMS) - 2

- **Erweiterungen für Multiprozessoren und abhängige Prozesse verfügbar**
- **asynchrone Prozesse mit maximaler Wiederholrate können in Analyse einbezogen werden (dann nicht optimal)**
- **Verwandtes Verfahren für Modell 2:**
  **Deadline Monotonic Scheduling (DMS)**:
  - Deadline liegt vor dem Ende der Periode
  - kürzere Deadline → höhere Priorität
  - *hier nicht weiter betrachtet*

## Rate Monotonic Scheduling (RMS) - 3

- **Theorem 1 (Liu/Layland 73):**
  Ein System von n unabhängigen Tasks, deren Priorität durch RMS bestimmt ist, wird immer dann alle Deadlines erreichen, wenn gilt (hinreichend):

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \quad \ldots \quad + \frac{C_n}{T_n} = U(n) \leq n(2^{1/n} - 1)$$

- wobei: $C_i$, $T_i$ Laufzeit und Periode des Tasks i

  lim (U(n)) = ln 2 = 0,69 U: Utilization
  $n \to \infty$



## Rate Monotonic Scheduling (RMS) - 4

- Beispiel 1
  P1: C1 = 20us, T1 = 100us $U1 = 0{,}20$
  P2: C2 = 40us, T2 = 150us $U2 = 0{,}27$
  P3: C3 = 100us, T3 = 350us $U3 = 0{,}29$

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \quad \ldots \quad + \frac{C_n}{T_n} = U(n) \leq n(2^{1/n} - 1) \quad ?$$

$$0{,}2 + 0{,}27 + 0{,}29 = 0{,}76$$

$$3(2^{1/3} - 1) = 0{,}779$$

$$0{,}76 \leq 0{,}779 \quad \Rightarrow \textit{Bedingung erfüllt}$$

Änderung: C1 = 40us $\Rightarrow$ U1 = 0,4

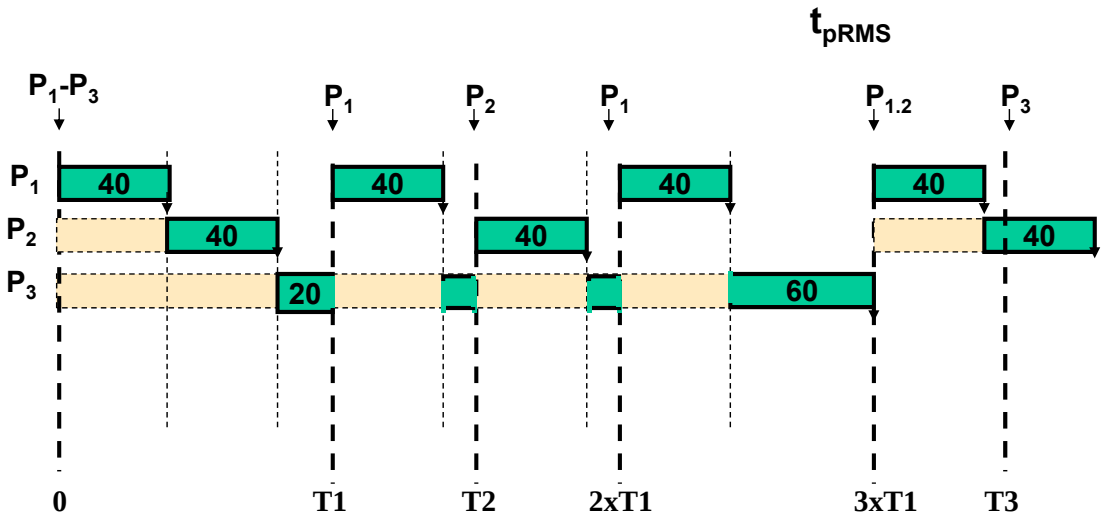
0,96 > 0,779 $\Rightarrow$ Bedingung nicht erfüllt, Schedule nicht garantiert

## Rate Monotonic Scheduling (RMS) - 5

- **Theorem 2 (Liu/Layland 73):**
  - Gegeben eine Menge von n unabhängigen Tasks, deren Priorität durch RMS bestimmt ist, und die zur selben Zeit aktiviert werden
    ***- critical instant***
  - Wenn jede Task ihre erste Deadline erreicht, dann werden alle künftigen Deadlines immer und für jede Kombination von Startzeiten eingehalten.
  - *notwendig und hinreichend (für $C_i = WCET_i$)*



## RMS Scheduling Beispiel




**alle Deadlines erreicht!**

- Beispiel 1a
  P1: C1 = 40us, T1 = 100us
  P2: C2 = 40us, T2 = 150us
  P3: C3 = 100us, T3 = 350us

## Rate Monotonic Scheduling (RMS) - 5

- **Beobachtung: Zu jeder Zeit t beträgt die gesamte *angeforderte Ressourcenzeit* (Service)**

$$W_n(t) = C_1\left\lceil \frac{t}{T_1} \right\rceil + C_2\left\lceil \frac{t}{T_2} \right\rceil + \quad \ldots \quad + C_n\left\lceil \frac{t}{T_n} \right\rceil$$

- **Im Beispiel:**
  - $W_3(0+)$ = C1 + C2 + C3 / alle drei Tasks erstmals aktiviert
  - $W_3(T1+)$ = 2*C1 + C2 + C3 / Task P1 zum zweiten Mal aktiviert
  - $W_3(T2+)$ = 2*C1 + 2*C2 + C3 / Task P2 zum zweiten Mal aktiviert
  - …
- **die angeforderten Ressourcen wurden zum Zeitpunkt t vollständig bereitgestellt, wenn gilt**

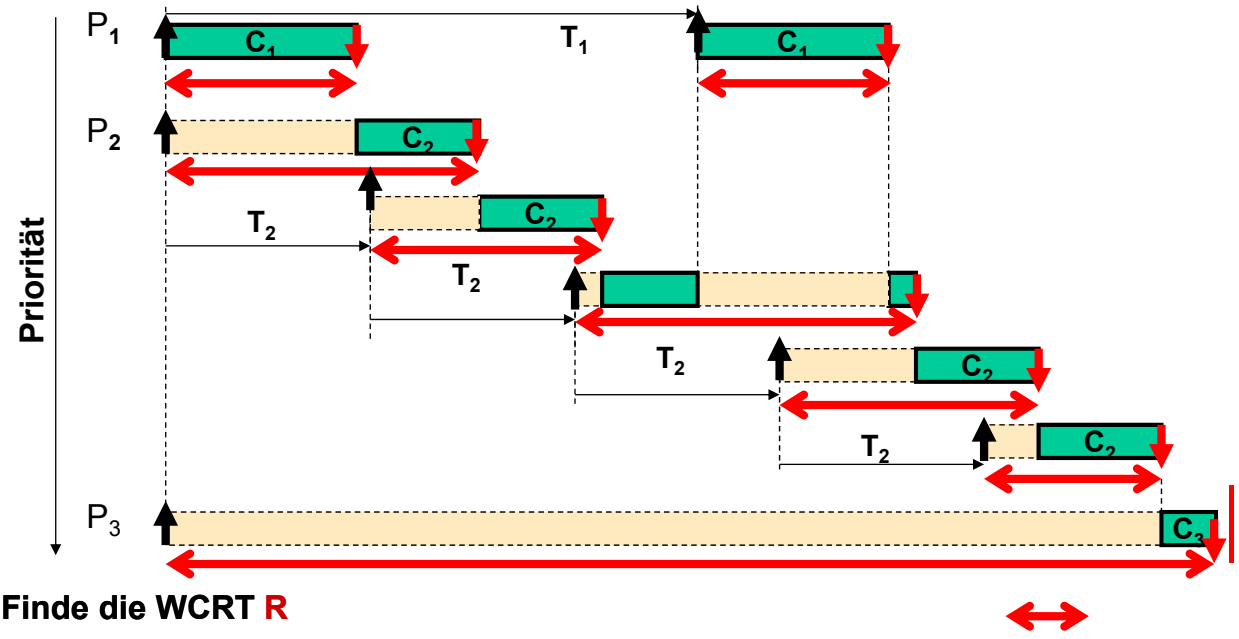
$$t = W_n(t)$$

**ausgehend vom Critical Instant müssen wir nur den ersten Zeitpunkt finden, wo die Gleichung gilt (Fixpunktproblem) → „Busy Window“**



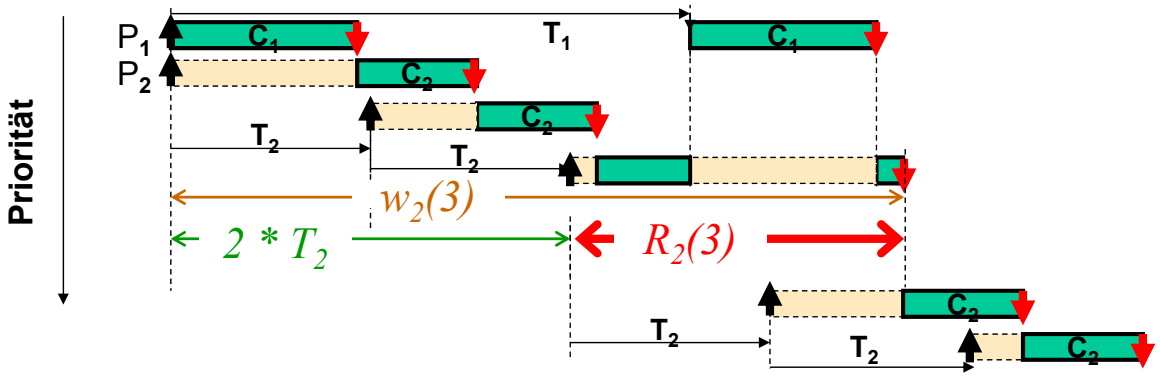
## Model 3: Beliebige Deadlines

**Analyse verallgemeinert das „Busy Window“- Verfahren (auch „Busy Period“)**




**Finde die WCRT R**

## Bestimmung der WCRT von P2




$$w_i(q) = \quad q\,C_i \quad + \sum_{j \in \mathrm{hp}(i)} C_j \left\lceil \frac{w_i(q)}{T_j} \right\rceil$$

**wieder Fixpunkt-gleichung**

$$R_i(q) = \quad w_i(q) \quad - \quad (q-1)\,T_i$$

**suche max. $R_2$ für alle q**

**hier: $R_2(3)$**



## Static Priority - Prioritätsinversion

- **Priority Inversion Problem**
  - tritt bei gegenseitigem Ausschluß z.B. bei Zugriff auf gemeinsame Daten oder Ressourcen (Interfaces) auf
  - Task P1 niederer Priorität kann dann höherprioren Task P2 für eine unbestimmte Zeit verzögern
- **Beispiel für invertierte Priorität:**
  - P1 höhere Priorität als P2 höhere Priorität als P3
  - P3 greift auf Semaphore S zu, blockiert S und tritt dann in einen kritischen Bereich ein
  - P1 wird bereit, wird ausgeführt und greift auf Semaphore S zu. Da S blockiert ist, wartet P1.
  - P2 wird bereit. Da P2 höhere Priorität als P3 besitzt, wird P3 unterbrochen und P2 fortgesetzt usw.

  → *P1 verfehlt seine Periode*

## Priority Ceiling Protocol - PCP

- **Ansatz: Priority Ceiling Protocol (PCP)**
  - S erhält eine Priorität und zwar so hoch, wie der höchste auf S zugreifende Task
  - wenn ein Task auf S zugreift und blockiert, erhält er die Priorität von S
- **Theorem 4 (PCP):**
  Das PCP vermeidet Deadlocks. Ein höherpriorer Task kann höchstens ein Mal durch einen niederprioren Task blockiert werden.

$$R_i = C_i + \sum_{j \in hp(i)} C_j \cdot \left\lceil \frac{R_i}{T_j} \right\rceil + \max_{k \notin hp(i)} (B_k)$$

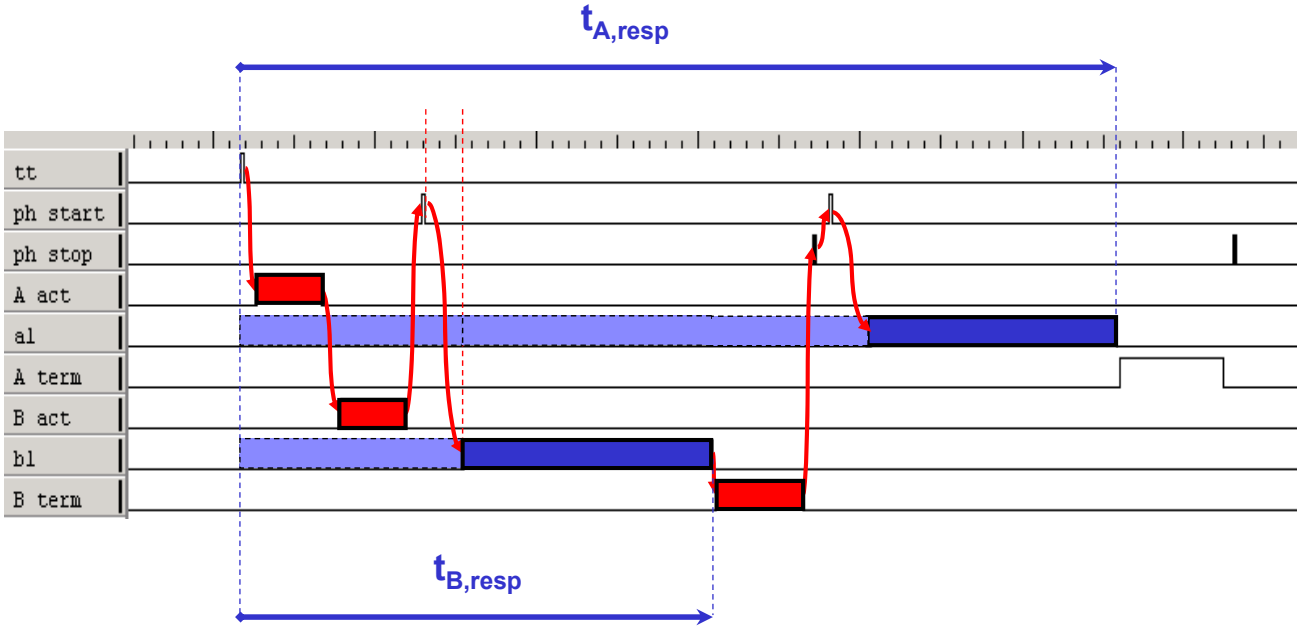
$$\forall i : R_i \leq T_i$$

- $R_i$ WCRT of task i; $T_i$ period (or minimum activation distance)
- $C_i$ WC execution time (WCET)
- hp(i) higher priority messages
- $B_k$: Blocking time
- für Deadline am Ende der Periode (q=1)



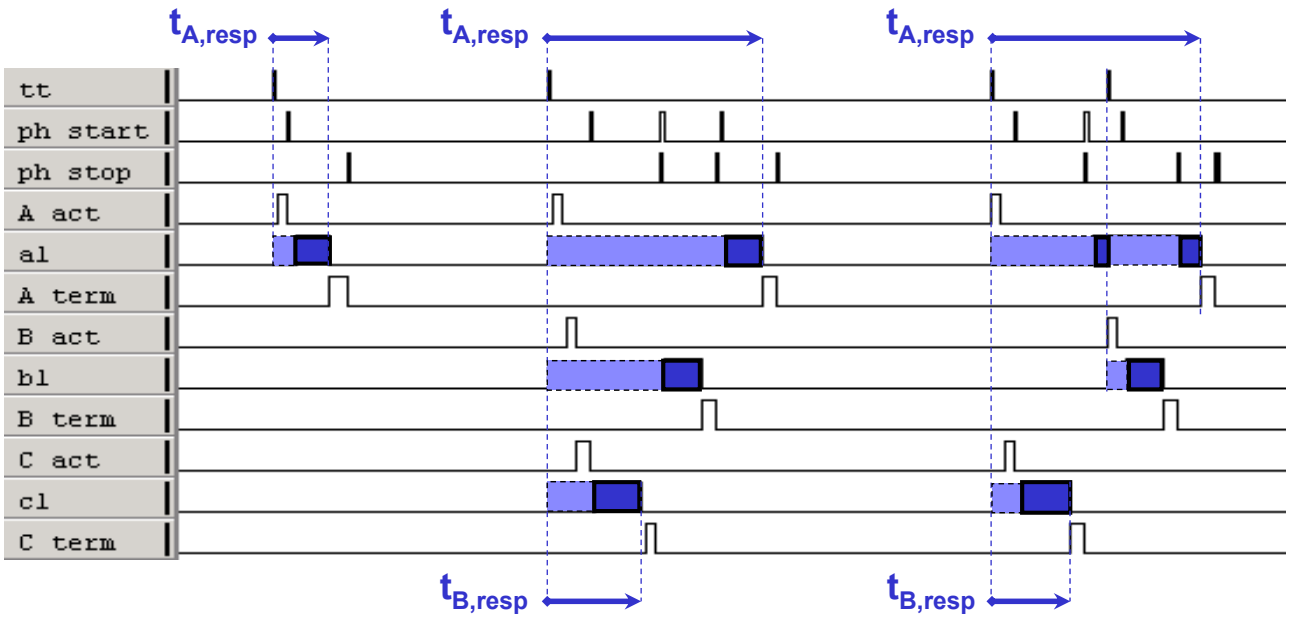
## RTOS Overhead

Example: Static priority scheduling with ERCOSEK™




**RTOS overhead increases response times**

## RTOS and scheduling effects combined



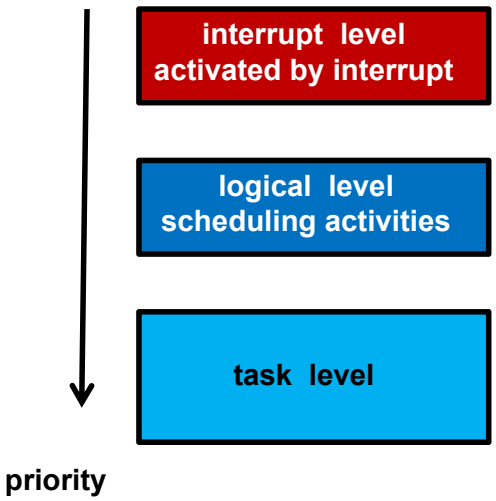



## Statische Prioritäten in der Praxis

- **Statische Prioritäten in der Kommunikation**
  - vergleichbar TDMA kennen alle Sender ihre Priorität im Voraus
  - keine Taktsynchronisation erforderlich
  - Zugriffsprotokoll benötigt
- **Statische Prioritäten können beliebig erweitert werden, solange freie Prioritäten verfügbar sind**
- **niemals Leerlauf, solange ein aktivierter Task vorliegt → lasterhaltend**
- **wichtige praktische Beispiele**
  - CAN
  - OSEK/AUTOSAR
  - (standard PC I/O-bus standards)

## Beispiele: Scheduling in OSEK/VDX und AUTOSAR

- **static priority preemptive scheduling (SPP)**
  - kann auf „Preemption Points“ beschränkt werden
- **3 Prioritätsblöcke**
  - interrupt – scheduling – task level
- **task level mit periodischen Tasks**
  - **Rate Monotonic Scheduling**
  - **Aktivierungs-Offsets** für verringerte Spitzenlast (verändert Critical Instant)
- **PCP Protokoll**




source: OSEK/VDX standard V2.2.3



## Dynamische Schedulingverfahren

- **Erreichen Auslastung bis U = 1 durch dynamische Anpassung der Prioritäten an die aktuellen Deadlines und Prozesse**
- **Ansatz im Einzelprozessor: Earliest Deadline First (EDF)**
  Der Prozeß wird ausgeführt, der die kürzeste Deadline besitzt
- **EDF erfordert Scheduling zur Laufzeit**
  (vgl. Rechnerarchitektur: dynamisches Befehlsscheduling)
- **bei bekannter Last lassen sich auch für EDF Schedulability Analysen durchführen**
  - spielt in der Praxis eingebetteter Systeme aber eine geringere Rolle als Verfahren mit statischen Prioritäten
  - beliebtes Verfahren in der Schedulingtheorie
    - gute Ansätze für Multicore-Systeme

# 5.4 Globales Systemverhalten und seine Analyse

- **bei periodisch aktivierten Tasks wird die Last im Modell durch die Komponente selbst bestimmt**
  - periodische Aktivierung – kein Einfluss von Ereignissen am Eingang
  - Ressourcennutzung wurde zu WCET oder den ungünstigsten Trace abstrahiert

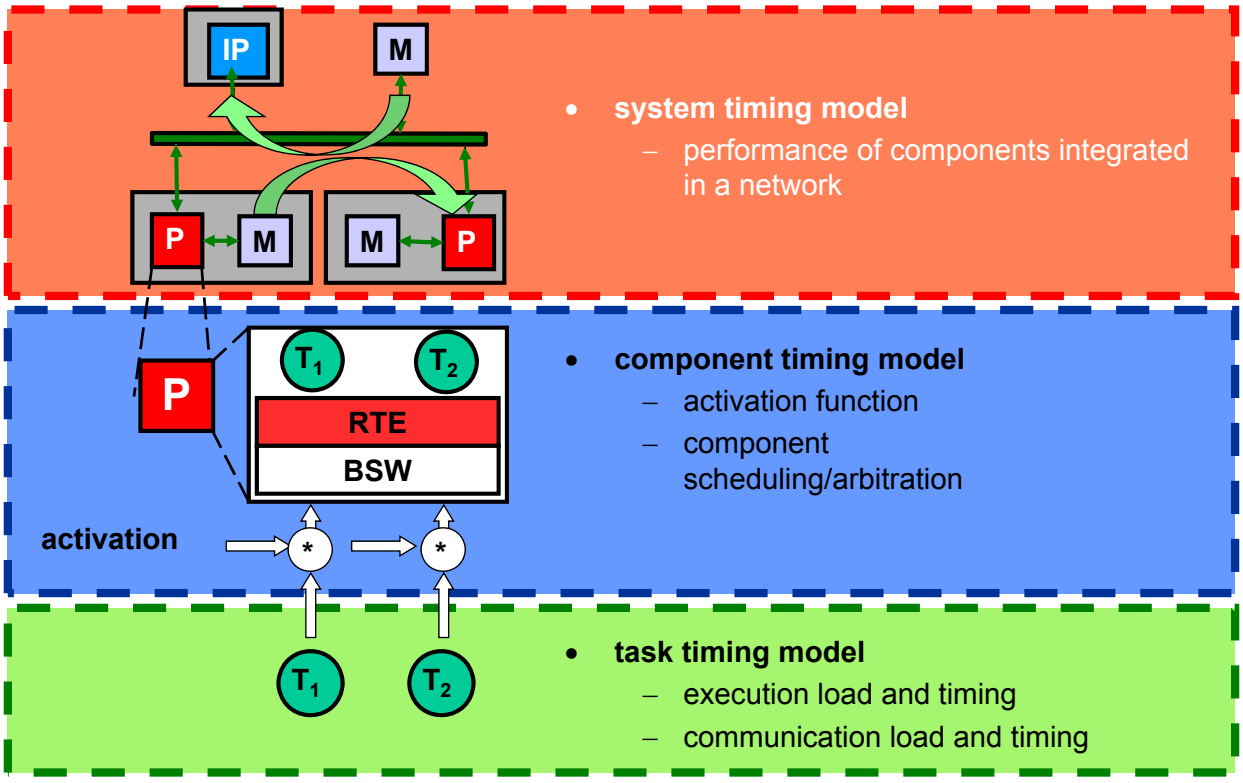
  → Effekte des Scheduling, z.B. WCRT oder $ET^{-/+}(n)$, können lokal bestimmt werden

  → die Scheduling-Effekte von Systemen mit *ausschließlich* periodischer Last sind **composable**
- **bei Aktivierung durch Ereignisse muss das Verhalten vorangehender Komponenten berücksichtigt werden**
  - erfordert Analyse auf Systemebene
  - typisch für reaktive Systeme und Kommunikationsnetze (daher Network Calculus)

  → die Scheduling-Effekte solcher Systeme sind **nicht composable**
  - **aber: es gibt Methoden zur Erzielung von Compositionality**

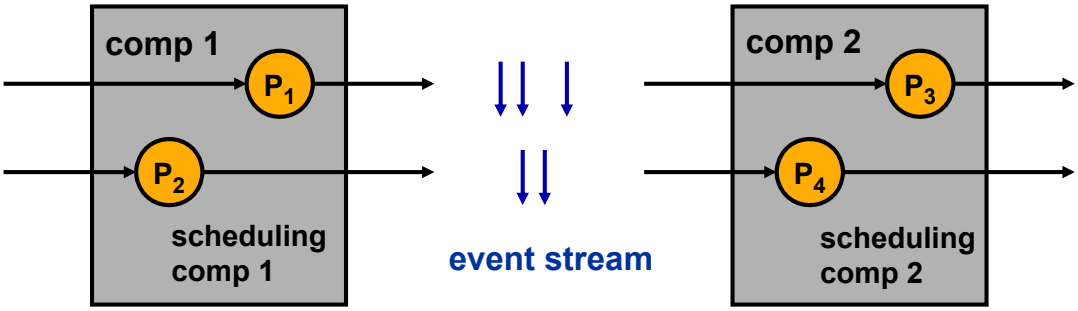


# Timing Model Hierarchy für globale Analyse



## Global system analysis using compositional approach

- independently scheduled subsystems are coupled by data flow



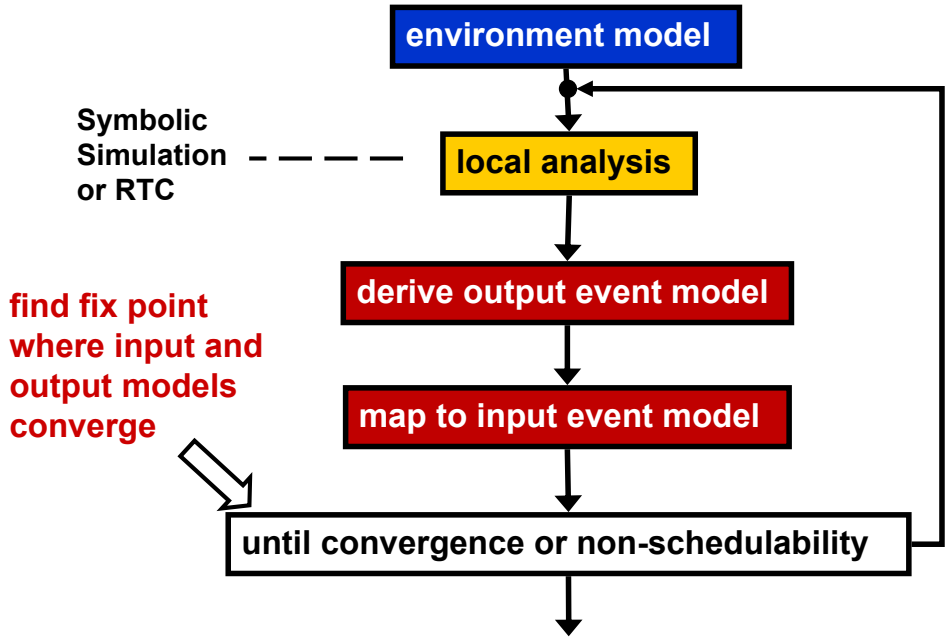

⇒ subsystems coupled by streams of data

⇒ interpreted as activating events

⇒ coupling corresponds to event propagation



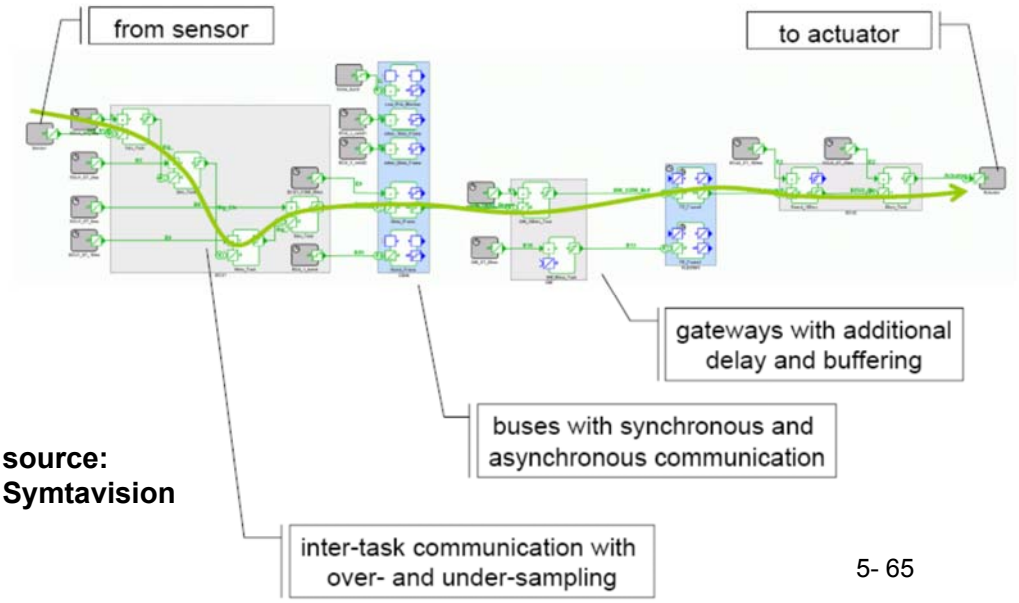
## Compositional analysis principle



## System-level Analysis Results

- **end-to-end latencies**
- **buffer sizes**
- **system load**
- **….**

**example: complex end-to-end latency analysis w. SymTA/S**

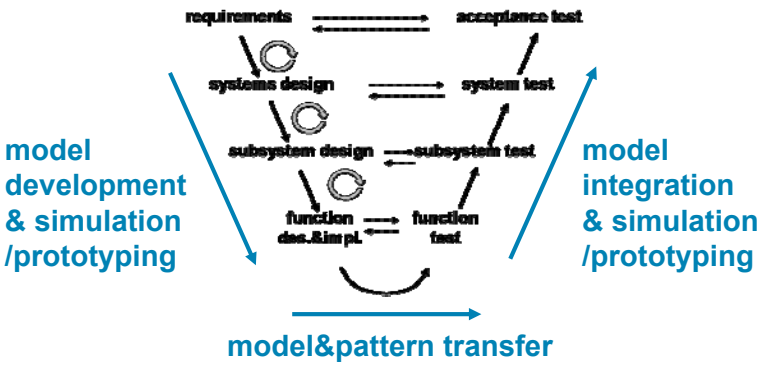



**source: Symtavision**



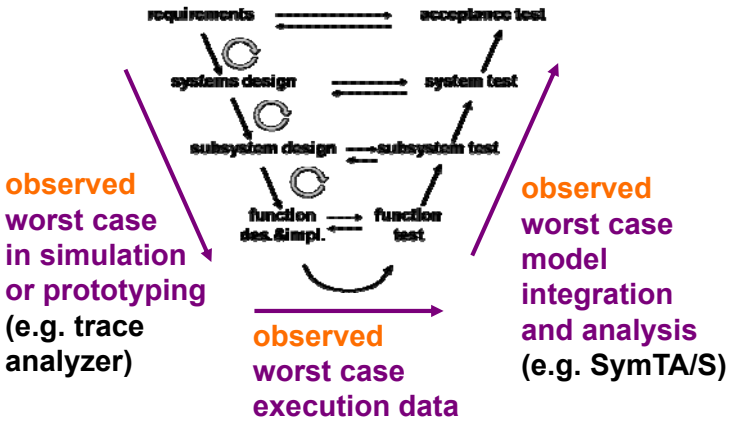
## Gewinnung der benötigten Daten

- **Test mit Simulation/Prototypen**
  - Simulation setzt Verfügbarkeit von Modellen voraus – oft nicht der Fall
  - Prototyping erst spät im Entwurf möglich oder mit Komponenten mit anderem Zeitverhalten
  - Anwendungen und Architekturen mit vielen zeitlichen Abhängigkeiten sind große Herausforderung und erfordern bei Änderungen Wiederholung des Testaufwands



## Gewinnung der benötigten Daten

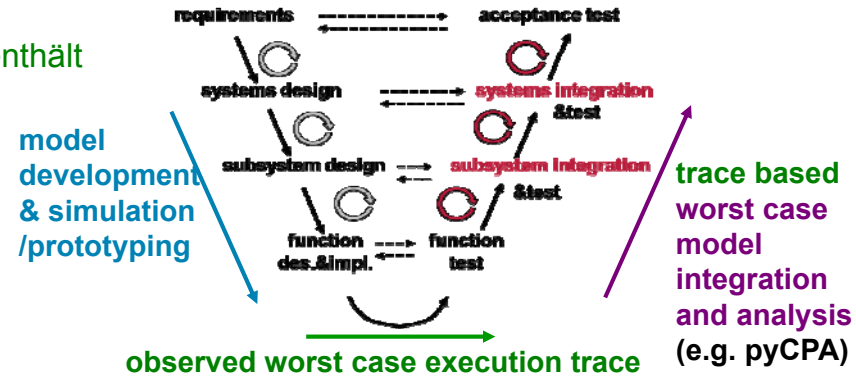
- **Formale Worst Case Modelle**
  - haupts. für kritische Anwendungen
  - genaue Modelle für analytische Verfahren (u.a. Fa. AbsInt) fehlen oft
- **Observed Worst Case Modelle**
  - **Messung** der WCET im Prototypen oder auf einem Simulator
  - deutlich höhere Zuverlässigkeit der Messung als im komplexen Systemtest






## Daten aus Traces – Trace Based Analysis

- **für genauere Workload-Modelle werden Traces benötigt**
  - **Messung von Worst Case Execution Traces**
- **formale Methoden basierend auf dem Busy-Window-Verfahren und der globalen Analyse verfügbar**
  - nutzt worst case execution trace (formal oder observed)
  - wesentlich weniger konservativ als die Worst Case Analysis
- **guaranteed behavioral bounds**
  - sofern der Trace den Worst Case enthält



# 5.5 End-zu-End Latenzen und Wirkketten

Verbliebene Frage:

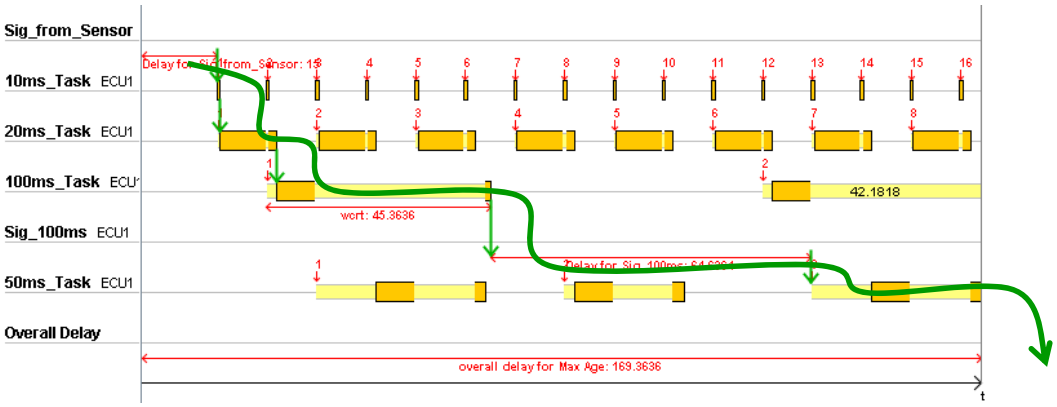
- **Beeinflusst das Zeitverhalten die Funktion?**
  - werden globale Deadlines der Anwendung erreicht?
  - wird die Funktion durch das Zeitverhalten beeinflusst?
- **unterschiedliche Effekte bei Zeit- und Ereignisaktivierung**
- **Ereignisaktivierung**
  - aus der Response Time lässt sich die Latenz pro Komponente bestimmen
  - die Summe aller Komponentenlatenzen ergibt die Gesamtlatenz
  - dabei können Sondereffekte genutzt werden („Pay Burst Only Once“)

  → **compositional**



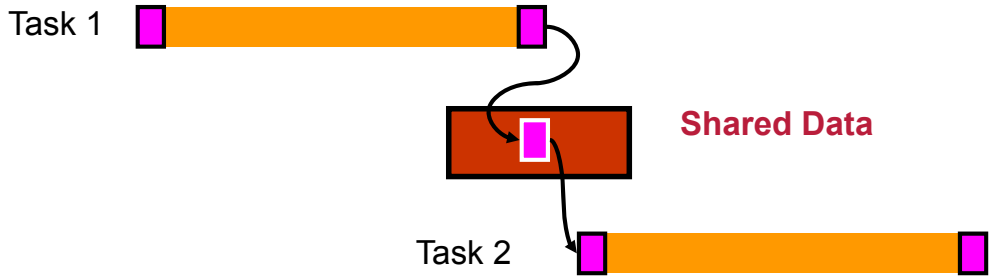
## Wirkketten in zeitgesteuerten Systemen

- **Ende-zu-Ende-Verhalten in zeitgesteuerten Systemen ist deutlich komplizierter**
  - Registerkommunikation führt zu Überschreiben von Werten
  - Überschreiben und Lesen ist abhängig vom **Lese- und Schreibzeitpunkt** – abhängig von den **Antwortzeiten der Komponenten**



## Kommunikation bei zeitaktivierten Tasks

- **typische Vorschrift für Datenkonsistenz**
  - **Lesen der kommunizierten Daten am Anfang einer Task**
  - **Schreiben der kommunizierten Daten am Ende der Task**
  - **atomarer Ablauf: oft mit Locks (Semaphore) geschützt – siehe PCP Protokoll**

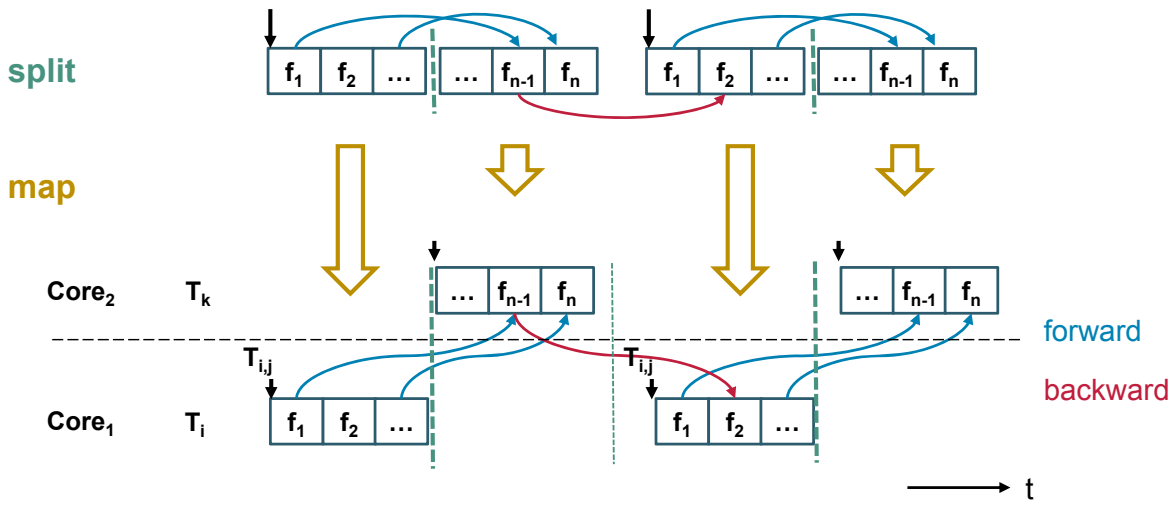



- **Kommunikation zeitabhängig**
  - wird ein Task früher/später beendet, kann ein früherer/späterer Wert gelesen werden
  - erfordert **Wirkkettenanalyse** - **compositional**
  - ändert sich mit dem Softwarestand, dem verwendeten Prozessor, …



## Wirkketten und Multicore-Architekturen

- **in Multicore-Architekturen entstehen komplizierte Zeiteffekte**
  - durch Zerlegung von großen Container-Tasks
  - durch Parallelverarbeitung
- **erfordert Synchronisation der Runnables $f_i$**



## Forderung Synchronisation der Kommunikation

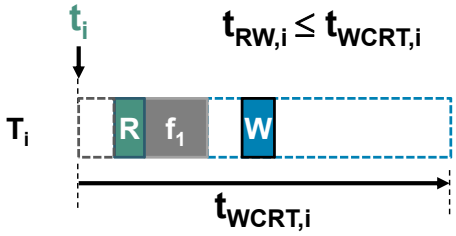
- **die übliche Synchronization von Zugriffen über Mutex-Variable führt zu Blockierungseffekten (z.B. mit Multicore-Erweiterung von PCP), die Jitter und Antwortzeit erhöhen (Diss. Negrean)**
- **stattdessen wird eine nicht blockierende (lock-free) Synchronization der Zugriffe über feste Zeitmarken angestrebt**
  - die Kommunikationszeitpunkte werden Teil des Implementierungsmodells



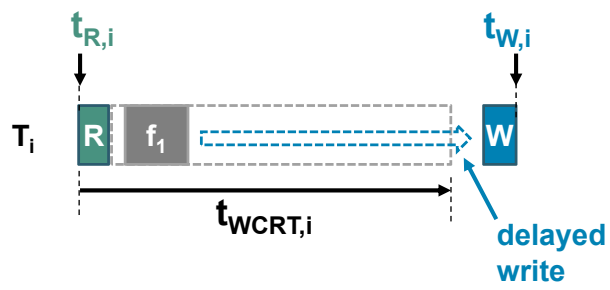
## Wichtiges Verfahren für zeitgenaue Kommunikation: LET

- **„Logical Execution Time“ Paradigma (LET)**
  - Trennung von Berechnung und Kommunikation
  - kommuniziere nur zu **festen Zeitpunkten** – **LET Marken**
  - kommunizierte Werte sind **unverzüglich für alle Tasks verfügbar $\Delta t = 0$** (vgl. synchrone Automaten)

übliche Taskausführung:
**Bounded Execution Time (BET)**
*aktiviert* zum Zeitpunkt $t_i$




Taskausführung mit
**Logical Execution Time (LET)**
*Lesevorgang* **exakt** zum Zeitpunkt $t_{R,i}$
*Schreibvorgang* **exakt** zum Zeitpunkt $t_{W,i}$



## Bewertung LET

- **LET führt zu deterministischer Kommunikation wie im Fall von synchronen Automaten**
- **LET macht das Zeitverhalten sogar composable**
  - gut für inkrementellen Entwurf, Reuse oder Updates
- **LET ist lasterhaltend**
- **die Forderung $\Delta t = 0$ für die Kommunikation ist so nicht erfüllbar**
  - praktische Umsetzung z.B. über Doppelpuffer, nicht ganz einfach bei Multicore
- **LET muss maximale Latenzen annehmen**
  - composability geht auf Kosten der erreichbaren Ende-zu-Ende Latenzen

  → Anpassung der LET Marken an die Architektur

  → spezielle Schedulingverfahren mit dynamischer Priorität („Priority Boosting“)
- **erste Multicore-Implementierungen mit LET in der Industrie**
- **aktives Forschungsthema für den Systementwurf**



## Zusammenfassung

- **Kap. 5 hat nur einen kleinen Einblick in die vielfältigen Modelle und Schedulingstrategien der eingebetteten Systeme geben können**
- **die Ereignismodelle für die Schedulingverfahren sind kompatibel zu den Modellen der Funktionsarchitektur**
- **die systematische Integration von Funktionen auf einer Ausführungsplattform erfolgt heute zunehmend unter Verwendung dieser formalen Modelle**
  - erhöht die Sicherheit im Entwurf, vor allem bei Multicore
  - neuere Architekturen machen die zeitliche Vorhersage aber schwieriger
  - formale Modelle ermöglichen die Ableitung von Monitoren zur Laufzeitüberwachung (s.a. Kapitel 6)
- **Forschungsthemen liegen unter anderem in neuen Modellen für probabilistische und weakly-hard Systeme und in einer systematischen Abhängigkeitsanalyse**
- **im Prinzip wäre mit den Methoden auch eine automatisierte Abbildung von Funktionen auf Plattformen möglich**
  - ist aber (noch) nicht Stand der Technik