ChatGPT

Lernskript Echtzeitsysteme – Klausurvorbereitung

Zeit und Ordnung (Time and Order)

In verteilten Echtzeitsystemen spielen sowohl **kausale** als auch **zeitliche Ordnung** von Ereignissen eine Rolle. Kausale Ordnung bedeutet, dass ein Ereignis *e1* nur Ursache von *e2* sein kann, wenn *e1* zeitlich vor *e2* stattgefunden hat [1]. Zeitliche (temporale) Ordnung vergleicht lediglich die Zeitpunktreihenfolge (wer war zuerst), ohne einen Ursachenbezug herzustellen [2]. Wichtig ist: Kausale Ordnung impliziert immer die zeitliche Ordnung, aber nicht umgekehrt [1]. (Beispiel: Wenn Person A in Wien und Person B in Los Angeles parallel Tickets buchen, entscheidet die globale Zeit, wer *wirklich* zuerst war – lokale Zeiten allein genügen nicht.)

Ohne globale physikalische Zeit greifen verteilte Systeme auf **logische Uhren** zurück. *Lamport-Uhren* ordnen Ereignissen monoton wachsende Zählerwerte zu, so dass für alle kausal abhängigen Ereignisse die Zeitstempel-Regel gilt: wenn $a \rightarrow b$ (a verursacht b), dann $C(a) < C(b)$ [3]. Damit ist die Konsistenz gewährleistet (`happened-before`-Relation), allerdings entsteht nur eine totale Reihenfolge, keine echte Kausalität. *Vektoruhren* verfeinern dies, indem jeder Knoten einen Zeitstempel-Vektor führt – so lässt sich auch erkennen, ob Ereignisse parallel (unabhängig) sind. Beide Verfahren (Lamport, Vector Time) sind Beispiele für **logische Zeit** in verteilten Systemen [4], um ohne physische Uhr eine geordnete Abfolge abzubilden. Darüber hinaus sind für Echtzeitbetrachtungen Begriffe wie **Ereignis** und **Dauer** relevant: Die Messung von Zeitintervallen ist stets mit einer gewissen Unsicherheit behaftet (Granularität der Uhren). Mit einer globalen Zeit lassen sich Intervallgrenzen jeweils bis auf ±g genau bestimmen; daraus ergibt sich für die gemessene Dauern eine Unsicherheit von ±2g [5] (wobei *g* die Granularität in Sekunden ist). Je nach benötigter Genauigkeit muss diese Unschärfe berücksichtigt werden.

In einem **dichten Zeitmodell** (*dense timebase*) können Ereignisse zu beliebigen Zeitpunkten auftreten. Im Gegensatz dazu stehen **sparsame Zeitbasen** (*sparse time*): Hier ist die Zeit in aktivitäts- und stille Phasen unterteilt. Ein π/Δ-sparsches Zeitmodell erzwingt, dass Ereignisse nur in kurzen Aktivitätsintervallen π stattfinden, die von Phasen der Inaktivität Δ getrennt sind [6]. Diese *Sparse Time* erleichtert die eindeutige Ordnung: Finden zwei Ereignisse in unterschiedlichen Aktivitätsintervallen statt, kann man ihre Reihenfolge sicher bestimmen [7]. Mit genügend langer Ruhephase Δ (relativ zur Synchronisationsungenauigkeit) wird gewährleistet, dass gleichzeitige Ereignisse in verschiedenen Knoten nicht zu Ordnungsproblemen führen.

Da verteilte Systeme ohne Abstimmung abweichende lokale Uhren haben können, ist eine **globale Zeitbasis** unerlässlich. Eine globale Zeit in einem verteilten Echtzeitsystem muss bestimmte Anforderungen erfüllen: *chronoskopisches Verhalten* (Uhr läuft ohne Sprünge, auch während Synchronisation), *bekannte Präzision Π* (max. Abweichung zwischen Knotenuhren), *hohe Zuverlässigkeit* und die *Metrik einer physikalischen Sekunde* [8]. Um eine solche Zeitbasis bereitzustellen, synchronisiert man die lokalen Uhren regelmäßig. Bei der **internen Uhrensynchronisation** gleichen die Knoten ihre Uhren untereinander ab, um eine Präzision Π einzuhalten [9]. Bei der **externen Synchronisation** werden die Knoten zusätzlich an eine Referenzzeit (z.B. GPS) gekoppelt [10] [11]. Intern synchronisierte Uhren garantieren, dass für jedes Ereignis e die Zeitstempel in allen fehlerfreien Knoten höchstens $|t_j(e) - t_k(e)| \leq \Pi$ auseinander liegen. In der Praxis erreicht man dies durch

Synchronisationsalgorithmen: Beispielsweise sendet beim **Central-Master-Algorithmus** ein Master periodisch Zeitnachrichten an alle Slaves, die daraufhin ihre lokalen Uhren justieren [12]. Verteilt arbeitende Algorithmen (etwa durch Austausch von Zeitstempeln in Runden) können Ausreißer erkennen und mitteln. Eine Bedingung für korrektes Arbeiten ist die Synchronisationsbedingung $\Phi + \Gamma \leq \Pi$ [13], wobei Φ der maximale Fehler direkt nach einer Sync-Phase ist und Γ die Driftabweichung bis zur nächsten Phase (Φ ergibt sich aus Nachrichten-Jitter, $\Gamma = 2\rho \cdot R_{int}$ mit ρ als max. Drift der Uhren und R_{int} als Resync-Intervall) [13]. In Gegenwart byzantinischer Fehler benötigt ein verteilter Algorithmus mindestens $N \geq 3k+1$ Uhren für k fehlerhafte, um noch einen Konsenszeitwert bestimmen zu können [14]. Praktisch bedeutet das, eine ausreichende Redundanz und Fehlerfilterung einzuplanen. Zur **externen Synchronisation** wird oft ein Hierarchie-Ansatz genutzt (z.B. Stratums bei NTP oder GPS-Anbindung eines Masterknotens), da externe Referenzen sehr genau, aber evtl. weniger verfügbar sind [11].

> **Beispielaufgabe (Zeit und Ordnung):** Zwei verschiedene Knoten eines verteilten Systems beobachten dasselbe externe Ereignis E und versehen es mit lokalen Zeitstempeln; Knoten A mit $t_A(E)=10$, Knoten B mit $t_B(E)=11$ (in Millisekunden). Beide Uhren sind global synchronisiert mit einer bekannten Präzision (Granularität) von $g = 1\ ms$. Kann man bestimmen, welcher Knoten das Ereignis *tatsächlich* zuerst gesehen hat?
>
> **Lösung:** Nein – bei einer Zeitstempeldifferenz von 1 und einer Granularität von 1 ms ist die Reihenfolge ungewiss. Zwei korrekt synchronisierte Uhren können um bis zu g auseinander liegen. Man bräuchte eine Differenz von mindestens $2 \cdot g$, hier also 2 ms, um die zeitliche Ordnung sicher vom Zeitstempel abzuleiten [15]. In diesem Beispiel beträgt der Unterschied nur 1 ms, was innerhalb der Synchronisationsungenauigkeit liegt. Folglich lässt sich nicht eindeutig entscheiden, ob Knoten A oder B E zuerst detektiert hat – das Ereignis könnte in Wahrheit zeitgleich gewesen sein. (Allgemein gilt: Ist $|t_j(e1) - t_k(e2)| < 2g$, so kann keine zeitliche Reihenfolge garantiert werden.)

Echtzeitmodelle und -komponenten

Ein **Echtzeitsystem** verbindet ein digitales Rechnersystem mit der physikalischen Umwelt. Wichtige Begriffe dabei sind *RT-Entity*, *RT-Image* und *RT-Objekt*. Eine **Real-Time Entity (RT-Entität)** bezeichnet ein relevantes Objekt oder Vorgang in der Umgebung, den es abzubilden gilt (z.B. Temperatur an einem Sensor, Position eines Fahrzeugs). Solche Entitäten haben **statische Attribute** wie Name, Typ, Wertebereich und maximale Änderungsrate, sowie **dynamische Attribute** wie den aktuellen Zustand bzw. Messwert zu einem gegebenen Zeitpunkt [16]. Das Rechnersystem erfasst den Zustand einer RT-Entität in einem **Real-Time Image (RT-Image)**, also einem Datenobjekt, das den Wert der Entität zu einem bestimmten Zeitpunkt repräsentiert [17]. Ein RT-Image gilt als *temporally accurate*, wenn es den Zustand der RT-Entität bezüglich Wert **und** Zeit genügend genau abbildet [17]. Dieses Abbild kann entweder auf einer direkten Beobachtung (Messung) oder auf einer Zustandsschätzung beruhen [18]. Gespeichert wird ein solches Image in einem **RT-Objekt**, einer Art Container im Computersystem, der typischerweise auch einen Zeitstempel enthält und an dem bei Eintreten bestimmter Ereignisse (neue Daten, Ablauf von Timer) definierte Aktionen ausgelöst werden können.

Die grundlegenden Bausteine eines komplexen Echtzeitsystems sind **Komponenten**. Eine Echtzeit-Komponente ist eine modulare Hard- und Softwareeinheit mit wohldefiniertem Zweck (*service*) [19] [20]. Sie umfasst typischerweise einen eigenen Rechnerknoten (Mikrocontroller mit Software) einschließlich aller relevanten **Zustandsdaten**. Eine Komponente erbringt ihren Dienst über klar beschriebene **Schnittstellen**. Man unterscheidet dabei meist vier Arten von Interfaces [21] [22]: (1) **Real-Time Service**

Interface – auch *Linking Interface (LIF)* genannt – verbindet die Komponente mit anderen Komponenten im Verbund (Netzwerknachrichten). Hier werden die Ein- und Ausgangsnachrichten sowie deren Timing und Bedeutung festgelegt [23]. (2) **Configuration/Planning Interface (CP)** – dient zur Offline-Konfiguration der Komponente und zur Planungsintegration ins Gesamtsystem (häufig technologieunabhängig gehalten). (3) **Lokales Interface** zur direkten Umgebung – bei *offenen* Komponenten, die Sensoren/Aktoren an physikalischen Prozessen haben, existiert eine Schnittstelle zur Hardwarewelt. (4) **Diagnostic Interface** – für Test, Debugging und Wartung (auch *TDI* genannt). All diese Interfaces einer Komponente haben definierte funktionale **und** zeitliche Eigenschaften [20], d.h. es wird nicht nur spezifiziert *was* ausgetauscht wird, sondern auch *wann* oder *wie oft* (z.B. Perioden, Deadlines) und ggf. *unter welchen Bedingungen*.

Eine Echtzeitkomponente enthält intern eine Menge von **Tasks** (Prozessen/Threads), welche die Anwendungssoftware ausführen. Die Komponente kann als *Unit of Failure* betrachtet werden – innerhalb einer Komponente wird oft auf schwergewichtige Isolation verzichtet, um Effizienz zu steigern, während zwischen Komponenten klare Grenzen bestehen. Der **Zustand** einer Komponente lässt sich als Summe des Zustands all ihrer Aufgaben und Kommunikationskanäle betrachten. Wichtig sind dabei folgende Konzepte: Die **History State (H-State)** umfasst alle Informationen über den aktuellen *und* bisherigen Lauf einer Komponente, die nötig sind, um sie an einem bestimmten Punkt fortzusetzen [24]. Sie beinhaltet z.B. den Inhalt wichtiger Variablen, ausstehende Nachrichten, Timerstände etc. Die **Initial State (I-State)** ist der Initialisierungszustand (Startwert) einer Komponente oder eines Tasks. Eine besondere Rolle spielt der **Ground State (G-State)** – das ist ein definierter Grundzustand des Systems mit minimaler H-State, erreicht etwa dadurch, dass alle Tasks inaktiv sind und keine Nachrichten mehr unterwegs (alle Kanäle leer) sind [25]. Im G-State befindet sich das System quasi in Ruhe; dieser Zustand ist nützlich, um einen synchronen Neustart oder eine fehlerfreie Reintegration durchzuführen. Ein *stateless node* wäre ein Knoten, der zwischen zwei Ausführungen keine Verlaufshistorie speichern muss – praktisch entspricht das dem Erreichen eines G-State zwischen zwei Aktivitätsphasen [26]. Rein stateless zu bleiben ist oft schwierig, aber Systeme streben an, sich zumindest periodisch einem definierten Grundzustand zu nähern (z.B. zyklische Sanity Points).

> **Beispielaufgabe (Komponenten & Interfaces):** *„Welche Eigenschaften müssen beschrieben werden, um das Linking-Interface zwischen zwei Subsystemen eines Echtzeitsystems vollständig zu charakterisieren?“*
>
> **Lösung:** Damit zwei Echtzeitsubsysteme über ein **Linking-Interface** (Real-Time Service Interface) gekoppelt werden, muss dessen Spezifikation mindestens folgende Aspekte abdecken [23]:
> 1. **Eingangsnachrichten:** Welche Nachrichten empfängt das Subsystem? (Format/Typ und Wertebereich der Eingangsdaten)
> 2. **Ausgangsnachrichten:** Welche Nachrichten sendet das Subsystem? (Format/Typ der ausgehenden Daten)
> 3. **Zeitliche Eigenschaften:** Mit welcher Rate bzw. zu welchen Zeitpunkten werden die Nachrichten erwartet/gesendet? (Perioden, Jitter, Deadlines, Phasenlage usw.)
> 4. **Bedeutung/Semantik:** Welche Bedeutung haben die ausgetauschten Nachrichten? (z.B. „Temperatursensorwert in °C“ oder „Bremspedalstellung aktiv/inaktiv“) – so ist klar, *was* die Daten repräsentieren.

Kommunikationsmechanismen und -protokolle

In Echtzeitsystemen spielen Kommunikationsverfahren eine zentrale Rolle, um verteilte Komponenten zuverlässig und fristgerecht Daten austauschen zu lassen. Grundsätzlich unterscheidet man

zeitgesteuerte (TT) und **ereignisgesteuerte (ET)** Kommunikationsprotokolle [27]. Bei einem *Event-Triggered (ET)* Protokoll wird die Übertragung durch das Auftreten von Ereignissen ausgelöst – d.h. eine Nachricht wird z.B. genau dann gesendet, **wenn** Daten anfallen oder sich ändern. Im Gegensatz dazu senden *Time-Triggered (TT)* Protokolle Nachrichten zu **fest vorab geplanten Zeitpunkten**, getriggert durch das Fortschreiten einer globalen Zeit [28]. Bei TT-Systemen ist somit der Sendezeitpunkt jeder Nachricht *a priori* allen Teilnehmern bekannt. Ein Beispiel ist das zeitgesteuerte TTP, wo jede Nachricht im Zeitslotplan genau einem Slot zugeordnet ist. Dadurch sind Kollisionen ausgeschlossen; die maximale Übertragungsdauer ist deterministisch und nahezu identisch mit der durchschnittlichen Dauer [28] [29]. ET-Systeme wie etwa ein übliches Ethernet oder CAN-Bus hingegen lösen Übertragungen asynchron aus und müssen mögliche Kollisionen oder Konkurrenz um das Medium auflösen. Dies macht sie flexibler und oft reaktionsschneller im Durchschnitt, jedoch ist ihr Worst-Case-Verhalten schwieriger abzuschätzen (höheres Jitter und potenzielle Verzögerungen bei Lastspitzen). Modernere Ansätze wie *Rate-Constrained (RC)* Protokolle stellen eine Zwischenform dar – sie garantieren jedem Sender eine bestimmte Bandbreite (Rate), lassen aber zeitliche Freiheiten innerhalb dieser Begrenzung. In vielen verteilten RTS werden TT- und ET-Kommunikation kombiniert eingesetzt (z.B. in FlexRay gibt es statische TT-Zeitslots und dynamische ET-Segmente).

Ein fundamentales Prinzip für robuste Kommunikation ist das **Ende-zu-Ende-Protokoll**. Hierbei wird der Effekt einer gesendeten Nachricht am **Endpunkt** überprüft und rückgemeldet, um sicherzustellen, dass die beabsichtigte Wirkung tatsächlich eintritt [30]. Ein Beispiel: Ein Sensor-Knoten sendet eine Meldung „Temperatur zu hoch“ an einen Aktor-Knoten; dieser Aktor führt daraufhin eine Kühlaktion aus und sendet eine Bestätigung zurück. Durch dieses Handshake weiß der Sender, dass seine Nachricht nicht nur angekommen, sondern auch umgesetzt wurde. Der Vorteil eines End-to-End-Protokolls liegt in der höheren **Fehlererkennungsrate** – Ausfälle oder Verluste auf dem Weg werden eher erkannt, und es wird verhindert, dass stille Fehler unbemerkt bleiben [31]. In sicherheitskritischen Systemen (z.B. Flugsteuerung) sind End-to-End-Checks essenziell, um zu validieren, dass Kommandos tatsächlich am Ziel wirken (Prinzip der *Endpunkt-Kontrolle*).

Ein weiteres wichtiges Konzept ist das **Agreement-Protokoll** (Konsensprotokoll). In einem verteilten System mit Redundanz können mehrere Knoten dieselbe physikalische Größe beobachten (bspw. mehrere Sensoren für die gleiche Temperatur). Ein Agreement-Protokoll stellt sicher, dass sich alle fehlerfreien Knoten auf **eine konsistente Sicht** einigen – also zum Beispiel, welcher Wert als *der* Temperaturwert gilt und welcher Zeitstempel ihm zugeordnet wird [32]. Dies ist insbesondere nötig, wenn analoge Größen in digitale Werte überführt werden (*A/D-Umsetzung*): Mehrere Sensoren könnten dicht aufeinanderfolgende Analoge Werte in leicht unterschiedliche digitale Zahlen wandeln. Durch ein Konsensverfahren (z.B. Mittelwertbildung oder Auswahl des Mehrheitswertes mit zugehöriger Zeit) einigen sich die Knoten auf einen einzigen Repräsentanten dieses Ereignisses [33]. Agreement-Protokolle sind somit für Fehlertoleranz entscheidend – sie verhindern, dass unterschiedliche Knoten divergierende Entscheidungen treffen, obwohl sie dieselbe Umweltänderung beobachten.

Ein spezieller Mechanismus für konsistente Datenhaltung ohne Sperren ist das **Non-Blocking Write (NBW)** Protokoll. Dabei handelt es sich weniger um ein Kommunikationsprotokoll im Netzwerk, sondern um ein Synchronisationsverfahren für gemeinsam genutzte Daten (z.B. in einem Shared Memory oder einer Memory-Mapped Kommunikation zwischen Rechnerknoten). NBW erlaubt **einem Schreiber und mehreren Lesern**, gleichzeitig auf eine Datenstruktur zuzugreifen, ohne dass die Leser den Schreiber blockieren müssen [34] [35]. Realisiert wird dies durch ein **Concurrency Control Flag (CCF)** und einen Doppeleintrag: Bevor der Writer die gemeinsamen Daten aktualisiert, setzt er ein Flag auf einen Zwischenwert, schreibt dann die neuen Daten vollständig und markiert anschließend das Flag mit einem Endwert [36]. Die Leser prüfen während des Lesens das Flag zu Beginn und am Ende ihrer Leseoperation. Wenn sich das Flag währenddessen ändert, verwirft der Leser die gelesenen Werte und beginnt nochmal (weil in diesem Moment ein Schreibvorgang dazwischenkam) [37]. Diese Schleife

wiederholt ein Leser so lange, bis er konsistente Werte vorfindet. Durch dieses Protokoll wird garantiert, dass Lesevorgänge **konsistente Ergebnisse** erhalten und der **Writer nie warten** muss (keine Blockierung durch parallele Leser) [38]. Außerdem lässt sich zeigen, dass die Verzögerung für Leser im Worst-Case begrenzt ist (da das Flag nur zwei Zustände annehmen kann und spätestens nach einer Wiederholung stabil bleibt) [38]. NBW kommt z.B. in echtzeitfähigen Daten-Puffern oder Ring-Buffer-Implementierungen zum Einsatz, wo Lese-Zugriffe zeitlich unkritisch und Schreib-Zugriffe priorisiert sind.

> **Beispielaufgabe (Kommunikation):** *„Wie funktioniert die Medienzugriffs-Arbitration im CAN-Bus? Betrachten Sie zwei gleichzeitig sendende Knoten: Node A mit ID 0x300 und Node B mit ID 0x100."*
>
> **Lösung:** Der CAN-Bus nutzt **CSMA/CA** (Carrier Sense Multiple Access mit Collision Avoidance) als Zugriffsverfahren. Bei Sendekonflikten gewinnt die Nachricht mit der höheren Priorität, d.h. der niedrigeren numerischen ID. In unserem Beispiel konkurrieren ID 0x300 (A) und 0x100 (B); folglich sollte Node B gewinnen, da 0x100 < 0x300. Im Ablauf stellt sich das so dar: Beide Knoten senden bitweise ihren Identifier. Solange die Bits gleich sind, geht es weiter. An der ersten Stelle, an der sie unterschiedlich sind, sendet A eine `1` (rezessiv) während B eine `0` (dominant) sendet. Dieses dominante Bit setzt sich auf dem Bus durch. Node A erkennt, dass sein gesendetes Bit nicht dem Buspegel entspricht – also muss ein anderes Node mit höherer Priorität senden. Daraufhin bricht A sein Senden ab [39]. Node B hat die Arbitration gewonnen und darf sein Datenframe komplett übertragen. (Allgemein gilt: Bei CAN "gewinnt" der Sender mit der kleinsten ID, da dominante `0`-Bits logische Einsen überstimmen.) Die maximale Übertragungsrate im CAN ist durch dieses Verfahren und physikalische Grenzen limitiert – ca. 1 MBit/s auf 40 m Kabellänge. **Warum?** Jeder Sender muss warten, bis ein gesendetes Bit am *anderen* Busende ankommt, um festzustellen, ob es dominant bleibt [40]. Je länger die Leitung, desto größer diese Laufzeit; ab einer gewissen Bitrate würde ein Knoten das nächste Bit senden, bevor das vorherige das Ende erreicht – was zu Fehlern führt. Daher begrenzen Leitungslänge und Lichtgeschwindigkeit die Bitrate im CAN-System.

Taskmodelle und Betriebssystem-Aspekte

Die Software einer Echtzeit-Komponente ist üblicherweise in mehrere parallel ablaufende **Tasks** (Aufgaben, Threads oder Prozesse) gegliedert [41]. Das Echtzeit-Betriebssystem (RTOS) stellt für jeden Task eine Ausführungsumgebung bereit, sorgt für *Scheduling* (Ablaufplanung) und *Ressourcenmanagement* (z.B. Speicherschutz, Zeitverwaltung) und bietet Dienste wie Synchronisations- oder Kommunikations-Primitive. Bei **Hard Real-Time (HRT)** Systemen geht man von einer *Closed World Assumption* aus: alle Tasks, deren Timing (Perioden, Deadlines) und ihre Kommunikationsbeziehungen sind bereits zur Designzeit bekannt und geändert sich während des Laufs nicht [42]. Diese statische Sicht erlaubt es, das System vorab vollständig zu analysieren und deterministische Garantien über die Einhaltung von Deadlines zu geben [43]. **Soft Real-Time (SRT)** Systeme hingegen folgen einer *Open World Assumption*: hier können zur Laufzeit neue Tasks oder variable Lasten auftreten, die nicht alle im Voraus planbar sind [44]. Das OS muss dann Mechanismen zur Verfügung stellen, um Überlast zu handhaben (z.B. durch *best effort* Scheduling, Priorisierung nach Wichtigkeit, Qualitätsabstufungen). In SRT-Systemen wird oft mit *QoS (Quality of Service)* gearbeitet – wenn nicht alle Anforderungen erfüllt werden können, wird zumindest eine "beste mögliche" Güte angestrebt, anstatt harter Deadline-Erfüllung.

Ein Echtzeit-OS muss **Zeit- und Speichermanagement** besonders behandeln. HRT-Systeme verlangen in der Regel **räumliche und zeitliche Isolation** kritischer Tasks [45]. Das bedeutet z.B., dass ein kritischer Task nicht durch einen anderen verdrängt oder ausgebremst wird (zeitliche Isolation) und dass Speicherzugriffe voneinander geschützt sind (räumliche Isolation), soweit nötig. Allerdings werden HRT-Tasks oft **kooperativ** gestaltet, nicht kompetitiv [45] – d.h. sie wurden so entworfen, dass sie sich nicht gegenseitig blockieren, sondern nach einem festen Plan ablaufen. Dies erleichtert die Verifizierbarkeit. Man spricht von *light-weight OS*, wenn auf aufwendige Schutzmechanismen zwischen Tasks verzichtet wird, um Laufzeit-Overhead zu sparen [46] (z.B. kein Vollspeicherschutz zwischen Tasks innerhalb einer Komponente, da die Komponente als Ganzes betrachtet wird). Wichtig ist die Erkenntnis: In hochkritischen HRT-Systemen wird viel Aufwand in die **Entwurfsphase** verlagert – das System wird statisch ausgelegt, sodass zur Laufzeit möglichst wenig Entscheidungen offen sind.

Auf **Task-Ebene** unterscheidet man zwei grundlegende Taskmodelle: **Simple Tasks (S-Tasks)** und **Complex Tasks (C-Tasks)** [47] [48]. Ein **S-Task** läuft von seinem Start bis zu seinem Ende **ohne Unterbrechung oder interne Wartephasen** durch, vorausgesetzt er hat die CPU zur Verfügung [49]. Er führt innerhalb seines Task-Body keine blockierenden Operationen aus – d.h. keine explizite Synchronisation mit anderen Tasks, keine Empfangswarten auf Nachrichten, kein Schlafen auf Timer usw. [50]. Dadurch ist sein Ablauf *eigenständig* und vorab planbar. Alle benötigten **Input-Daten liegen zu Task-Beginn bereit**, z.B. aus einem Eingabepuffer oder Sensorwert, und sämtliche **Ausgaben sind am Task-Ende fertig** und werden dann in die Ausgabestruktur geschrieben [51]. Das bedeutet, S-Tasks haben einen klaren Anfang und Ende und hängen innerhalb nicht von externen Events ab. Viele zeitgesteuerte Systeme beschränken sich daher auf S-Tasks, da diese einfacher deterministisch zu behandeln sind. Dem gegenüber kann ein **C-Task** (komplexer Task) **eine oder mehrere Wartepunkte (WAITs)** innerhalb seines Ablaufs haben [48]. Beispielsweise könnte ein Task an einer Stelle auf ein Signal oder eine Nachricht eines anderen Tasks warten, bevor er fortfährt. Dadurch **ist sein Fortschritt nicht mehr unabhängig**, sondern kann von anderen Tasks oder der Umwelt abhängen [52]. Das macht sein zeitliches Verhalten zu einem *globalen* Problem – der Scheduler muss zur Laufzeit entscheiden, wann er wieder aktiviert wird, und es können Blockierungen auftreten. Ein C-Task kann also Synchronisationsmechanismen (wie Semaphore, Events) oder geteilte Datenstrukturen nutzen, was in der Programmierung mehr Sorgfalt erfordert. In der API eines C-Tasks kommen zusätzlich zu Eingabe-, Ausgabe- und ggf. globalem Zustand auch **Shared Data Structures** ins Spiel, die mit anderen Tasks geteilt werden [52]. Insgesamt sind C-Tasks flexibler, aber schwieriger hinsichtlich Worst-Case-Timing zu analysieren, während S-Tasks einfachere Garantien erlauben.

Ein Realzeitbetriebssystem verwaltet Tasks typischerweise in verschiedenen **Zuständen**: *aktiv/laufend* (der Task wird gerade auf der CPU ausgeführt), *bereit* (er könnte laufen, wartet aber auf CPU-Zuteilung durch den Scheduler), *blockiert/wartend* (er pausiert, weil er auf ein Ereignis oder eine Ressource wartet) und *beendet*. Der Scheduler wechselt Tasks von einem Zustand in den anderen, z.B. wenn ein Task seine CPU-Zeit verbraucht hat oder ein höher-priorer Task bereit wird (Preemption). Diese klassischen Zustandsautomaten gelten auch in Echtzeitsystemen, nur dass hier Preemption und Prioritäten besonders gestaltet werden (siehe Scheduling-Algorithmen unten). Außerdem gibt es den Zustand *schlafend* oder *suspendiert* in manchen OS, etwa wenn ein Task zeitlich geplant ist, aber noch nicht dran ist. – In streng zeitgesteuerten Systemen versuchen Designer allerdings, Blockadezustände zu minimieren, indem Tasks entkoppelt werden (z.B. nur S-Tasks verwenden und Kommunikation zeitgeplant machen).

> **Beispielaufgabe (Task-Typen):** Ein Task liest periodisch Sensordaten ein, verarbeitet sie und schreibt ein Ergebnis, **ohne** währenddessen auf externe Ereignisse zu warten oder zu blockieren. Handelt es sich um einen S-Task oder einen C-Task? Begründen Sie kurz.

Lösung: Das beschriebene Task-Verhalten entspricht einem **Simple Task (S-Task)**. Ein S-Task läuft von Anfang bis Ende durch, sobald die CPU ihm zugeteilt wurde [49]. In diesem Szenario gibt es keine Synchronisationspunkte oder Wartezeiten innerhalb des Tasks – er hat einen unabhängigen sequentiellen Ablauf. Alle Eingaben (neue Sensordaten) liegen zu Taskbeginn vor, Ausgaben werden am Ende geschrieben. Es tritt also kein *Blocking* im Task-Body auf, was genau der Definition eines S-Tasks entspricht [50]. Ein C-Task wäre es nur, wenn z.B. der Task mittendrin auf eine externe Nachricht warten müsste oder durch Synchronisation mit anderen Tasks pausieren würde [48], was hier nicht der Fall ist. (Insbesondere sind periodische Lese/Schreib-Aufgaben typisch als S-Tasks realisierbar.)

Zeitgesteuerte vs. ereignisgesteuerte Systeme

Ein wesentliches Unterscheidungsmerkmal von Echtzeitsystemen ist die Art, wie die **Ablaufsteuerung** erfolgt: Zeitgesteuert (TT) oder ereignisgesteuert (ET). Dieses Prinzip betrifft vor allem das Scheduling im Betriebssystem und die Auslösung von Tasks.

In einem **zeitgesteuerten (TT)** System sind *sämtliche Aktivitäten im Voraus zeitlich geplant*. Das bedeutet, es existiert ein statischer **Zeitplan** (Schedule), der während der Designphase mittels Offline-Scheduling erstellt wurde. Dieser Plan umfasst z.B. eine Tabelle aller Task-Aktivierungen, Nachrichtenversendungen etc. mit genauen Zeitpunkten. Zur Laufzeit sorgt dann ein **Dispatcher** dafür, dass die Aufgaben strikt nach diesem Plan gestartet werden [53]. Konkret interpretiert der Dispatcher etwa eine Task-Deskriptor-Liste (**TADL**, Task Descriptor List) und aktiviert zur vorgesehenen Zeit den entsprechenden Task bzw. startet eine Routine [53]. Da dieser Zeitplan fix ist, gibt es **kaum dynamisches Ressourcenmanagement** im System: die CPU-Zeiten sind fest verteilt, Speicher- und Puffer sind statisch allokiert, Synchronisation erfolgt implizit durch das zeitliche Design [54]. Ein zeitgesteuertes OS braucht folglich keine komplexen Laufzeit-Scheduler; es genügt ein einfacher Zyklus-Dispatcher mit Timer-Interrupt. **Vorteile** eines TT-Systems sind die hohe **Determinismus** und Vorhersagbarkeit – jedes Timing-Verhalten wurde schon vorab geprüft. Kritische Ressourcen können nicht konflikthaft gleichzeitig angefordert werden, da Zugriffe entweder im Plan entkoppelt sind oder durch die Abfolge (z.B. keine zwei Tasks gleichzeitig auf einer CPU). Die **Closed-World-Annahme** ist hier Voraussetzung: das System geht davon aus, dass zur Laufzeit nichts Ungeplantes passiert (keine neuen Tasks, keine spontanen externe Jobs außer den vorgesehenen). Änderungen im Ablauf erfordern ein Neuberechnen des Zeitplans. Beispiele für TT-Systeme sind TTA (Time-Triggered Architecture) oder ARINC 653 Partitionen in Avionik, wo Intervalle streng aufgeteilt sind.

Demgegenüber steht das **ereignisgesteuerte (ET)** System. Hier werden Tasks und Aktionen aufgrund von *Ereignissen* geplant – z.B. das Eintreffen einer Nachricht, das Auslösen eines Sensors oder das Freiwerden einer Ressource. Das Betriebssystem eines ET-Systems enthält einen dynamischen **Scheduler**, der bei jedem relevanten Ereignis entscheidet, welcher Task als nächstes laufen soll. Meist kommen Prioritäten zum Einsatz: z.B. Ready-Tasks in eine Warteschlange, sortiert nach Priorität, und der mit höchster Priorität läuft (Preemptive Scheduling). **Vorteile**: ET-Systeme sind **flexibel** und können auf unvorhergesehene Ereignisse unmittelbar reagieren – etwa Interrupts sofort bedienen, neue Aufgaben einplanen etc. Sie funktionieren gut in offenen Welten, wo sich Lastsituationen ändern. Allerdings ist die Worst-Case-Analyse hier schwieriger: Bei ungünstigen Kombinationen von Ereignissen kann es zu Überlast kommen, Prioritätsinversionen können auftreten (siehe nächste Sektion), und allgemein ist **Determinismus** nur mit konservativen Annahmen zu erreichen. Ein bekanntes ET-Betriebssystem in Embedded-Systemen ist z.B. **OSEK** (bzw. AUTOSAR OS im Automotive-Bereich), das mehrere Prioritätsstufen und Ereignis-Hooks bietet, aber dessen exakte Timing-Analyse komplex ist (formale vollständige Verifizierung gilt als sehr herausfordernd). Oft versucht man, in ET-Systemen zumindest Teile statisch auszulegen oder Reservierungen einzubauen, um kritische Funktionen zu schützen. Insgesamt gilt: TT für sehr hohe zeitliche Strenge, ET für mehr Adaptivität. In der Praxis

existieren auch **hybride Systeme**, die eine Basis-Zeitsteuerung mit ereignisbedingten Zusatzaufgaben kombinieren – so lassen sich die Vorteile beider Ansätze verbinden.

Man kann die Unterschiede auch im **Ressourcenmanagement** sehen: In einem reinen zeitgesteuerten System werden CPU-Zuweisungen, Speicher und Kommunikationsfenster von vornherein festgelegt (keine Konkurrenz, keine Laufzeit-Allokatoren), während in einem ereignisgesteuerten System zur Laufzeit z.B. Speicher dynamisch alloziert werden kann oder die CPU auf wechselnde Tasks verteilt wird. Letzteres erfordert Synchronisationsmechanismen, ersteres nicht.

> **Beispielaufgabe (TT vs. ET):** *„Charakterisieren Sie das Ressourcenmanagement in einem strikt zeitgesteuerten Echtzeit-Betriebssystem."*
>
> **Lösung:** In einem zeitgesteuerten RTS-Betriebssystem gibt es nahezu **kein dynamisches Ressourcenmanagement zur Laufzeit** [55]. Konkret: Die CPU-Zeit wird statisch auf Tasks verteilt (fester Zeitplan statt eines Prioritätenschedulers) [56]. Das Speichermanagement erfolgt weitgehend autonom und benötigt kaum Eingriffe des OS, weil alle Speicherbedarfe vorher bekannt sind [57]. Buffer und Warteschlangen werden minimiert – idealerweise entstehen keine Wartequeues, da die Synchronisation implizit durch den Zeitplan sichergestellt ist [58]. **Synchronisations- und Kommunikationsanforderungen** werden durch die vorgeplanten Abläufe erfüllt, ohne dass zur Laufzeit Semaphore o.ä. nötig wären [59]. Oft beschränkt man sich auf **S-Tasks** (statt C-Tasks) [60], damit keine unvorhergesehenen Wartezeiten auftreten. Durch diese Maßnahmen bleibt das OS sehr schlank und deterministisch – es kann sogar formell verifiziert werden, da keine komplexen Scheduler-Entscheidungen oder Allokatoren berücksichtigt werden müssen [60]. Der Nachteil ist natürlich die geringe Flexibilität, aber für sicherheitskritische Systeme ist diese statische Strikte oft gewünscht.

Scheduling-Algorithmen in Echtzeitsystemen

Die **Ablaufplanung (Scheduling)** regelt, welcher Task zu welchem Zeitpunkt Rechenzeit erhält. In Echtzeitsystemen müssen Scheduler Deadlines berücksichtigen und garantieren (bei HRT) oder zumindest möglichst einhalten (SRT). Es gibt statische und dynamische Scheduling-Ansätze, sowie zeitgesteuerte Tabelle vs. prioritätsgesteuerte Verfahren. Wir betrachten hier einige wichtige Algorithmen: **Zyklische Scheduler**, **Fixed-Priority (RMS)** und **Dynamic-Priority (EDF)**.

Bei einem **zyklischen Scheduler** (Cyclic Executive) werden alle periodischen Tasks in einen zeitlichen Ausführungsrahmen (Major Cycle) eingeplant und als Sequenz von Prozeduraufrufen realisiert [61]. Ein einfaches Beispiel: In einem Major Cycle von 20ms liegen z.B. vier Minor Cycles, in denen jeweils eine bestimmte Gruppe von Prozeduren nacheinander aufgerufen wird [61]. Der Scheduler ist hier einfach eine Endlosschleife, die jede Minor Cycle per Timer startet und darin fix kodiert z.B. `task_a(); task_b(); task_c(); ...` ausführt [62]. Diese Prozeduren entsprechen den Funktionalitäten der Tasks. Vorteile: einfache Implementierung (ohne Kontextwechsel innerhalb des Zyklus), minimale Laufzeit-Overheads und vollständig vorhersagbar. Allerdings ist diese Form unflexibel – sporadische Aufgaben oder wechselnde Lasten lassen sich schlecht integrieren, und die Major Cycle muss so gewählt sein, dass alle Deadlines eingehalten werden. Zudem teilen sich alle Prozeduren den gleichen Adressraum (keine Hardware-Isolation). Zyklische Executives eignen sich gut für sehr starre Systeme mit festem Taskset und harmonischen Perioden.

Weit verbreitet in Prioritäten-gesteuerten Systemen ist das **Rate-Monotonic Scheduling (RMS)**. RMS ist ein *statisches Prioritäten-Scheduling*: Jedem Task wird offline eine feste Priorität zugewiesen, und zwar in

umgekehrt proportionaler Relation zu seiner Periode (höhere Frequenz = höhere Priorität) [63]. D.h. der Task mit der kürzesten Periode erhält die höchste Priorität [64], der längste Periodentask die niedrigste. Lai et al. (1973) haben gezeigt, dass diese Zuordnung **optimal** ist für feste Prioritäten: Wenn irgendeine Prioritätenanordnung ein Taskset fristgerecht planen kann, dann wird es auch RMS schaffen [64]. Zur Laufzeit führt man RMS typischerweise *präemptiv* aus – immer wenn ein höherpriorer Task bereit wird, verdrängt er ggf. einen niedrigeren. Die zentrale Frage ist die **Schedulability**: Wann gilt ein Taskset als planbar? Dafür gibt es Tests. Ein einfacher **hinreichender** Test für RMS ist der **Liu-Layland Utilization Bound**: Eine Menge von *n* unabhängigen, präemptiven, periodischen Tasks mit Deadlines = Perioden ist garantiert schedulbar unter RMS, wenn

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \cdot (2^{1/n} - 1).$$

Für große *n* konvergiert dieser Ausdruck gegen $\ln(2) ≈ 0.693$ (69,3%) [65]. Bei *n = 3* z.B. liegt die Grenze bei ~0,779. Ist die *Gesamtauslastung* U unterhalb dieses Wertes, so werden alle Deadlines sicher eingehalten. Liegt U darüber, kann das Taskset unter RMS ggf. trotzdem noch schedulbar sein – der Test ist nur hinreichend, nicht notwendig. Ein **notwendiger** Test ist trivialerweise $U ≤ 1$ (100% Auslastung darf nicht überschritten werden) [65]. Für engere Analyse verwendet man die **Response-Time-Analyse**: Dabei wird iterativ für jeden Task berechnet, ob seine Worst-Case-Ausführungszeit plus Interferenzen durch höhere Prioritäten innerhalb seiner Deadline bleibt. Diese genaue Methode ist aber rechenaufwändiger. RMS lässt sich effizient implementieren, da Prioritäten konstant und bekannt sind. Typische Systeme mit RMS sind z.B. viele RTOS wie VxWorks (als Option) oder OSEK (mit statischer Prioritätentabelle).

Earliest Deadline First (EDF) ist ein *dynamischer Prioritäten-Algorithmus*. Dabei werden Prioritäten nicht fest vergeben, sondern **laufend nach Deadline** bestimmt: Der Task mit der kleinsten absoluten Deadline (d.h. der am dringendsten fälligen Aufgabe) bekommt jeweils die CPU [66]. EDF ist auf einem Einzelprozessor **optimal** im Sinne der Schedulbarkeit – wenn ein Taskset mit irgendeinem Algorithmus machbar ist, dann schafft EDF es auch (und umgekehrt, falls nicht, kann kein Algorithmus es schaffen). Für EDF gibt es einen sehr einfachen exakten Schedulability-Test: Die Aufgaben sind genau dann alle terminrechtzeitig planbar, **wenn** $U = \sum C_i/T_i \le 1$ [67]. Ist die Gesamtauslastung ≤100%, so findet EDF immer einen Zeitplan, der alle Deadlines einhält. Liegt U über 1, ist es offensichtlich unmöglich. Dieser Test ist gleichzeitig notwendig und hinreichend für EDF [67] – im Gegensatz zu RMS, wo die 69%-Grenze nur hinreichend war. **Vorteile** von EDF: hohe CPU-Ausnutzung (bis 100% theoretisch), flexibles Reagieren auf Deadline-Änderungen (wenn z.B. Tasks sporadisch mit unterschiedlichen Deadlines kommen). **Nachteile**: Implementation etwas komplexer – man muss die Ready-Queue nach Deadlines sortiert halten, bei jedem neuen Task bzw. Timer-Ereignis die nächste Deadline ermitteln [68]. Außerdem verhält sich EDF in **Überlastsituationen** nicht so robust: Wenn $U>1$ (auch nur kurzfristig), kann es zu einer *Deadline-Kettenreaktion* (Domino-Effekt) kommen, bei der viele Aufgaben sukzessive ihre Termine verpassen [69]. Ein Fixed-Priority-System wie RMS würde in Überlast dagegen "nur" die niedrigsten Prioritäten verhungern lassen, aber hohe Prioritäten weiterhin bedienen (zumindest ein definiertes Teilsystem bleibt funktional). In Hard-RT-Umgebungen bevorzugt man oft RMS wegen dieser Überlastkontrolle, während EDF mehr im Soft-RT und allgemeinen Scheduling (z.B. Linux CFS zur Approximation) vorkommt.

Zur **Überprüfung** der Planbarkeit (Schedulability) nutzt man je nach Algorithmus unterschiedliche Methoden. Für RMS haben wir die Utilization-Bound-Schätzung genannt. Für EDF ist $U≤1$ exakt. Generell kann man immer einen **konkreten Schedule konstruieren**, um die Planbarkeit zu prüfen: das sogenannte *Zeitdiagramm*. Insbesondere für wenige Tasks lässt sich die Zeitachse in Hyperperiode (kleinstes gemeinsames Vielfaches aller Perioden) abtragen und jeder Task startend bei Phase 0

eingezeichnet werden – graphisch erkennt man, ob Deadlineverletzungen auftreten. Diese Methode ist zwar nicht formal, aber anschaulich und oft ausreichend, wenn U hoch ist und Tests uneindeutig sind.

> **Beispielaufgabe (Schedulability):** Drei unabhängige periodische Tasks seien gegeben mit Perioden gleich ihren Deadlines:
> – T1: Periode 8 ms, Berechnungszeit C=4 ms
> – T2: Periode 16 ms, C=3 ms
> – T3: Periode 4 ms, C=1 ms
> Prüfen Sie, ob dieses Task-Set unter **Rate-Monotonic Scheduling (RMS)** fristgerecht ausgeführt werden kann. Falls nein, beurteilen Sie, ob **EDF** das Set schedulen könnte.
>
> **Lösung:** Zunächst rechnen wir die Gesamtauslastung U aus:
> $U = \frac{4}{8} + \frac{3}{16} + \frac{1}{4} = 0.5 + 0.1875 + 0.25 = 0.9375 = 93.75\%.$ [70]
> Für *n=3* Tasks liegt der **RMS-Grenzwert** bei $3 \cdot (2^{1/3} - 1) ≈ 3 \cdot (1.26 - 1) = 0.78$ bzw. 78% [71]. Da $U = 0.9375$ diesen Wert deutlich **überschreitet**, fällt der hinreichende Schedulability-Test negativ aus – das System ist *nicht garantiert* mit RMS planbar [72]. Tatsächlich würde in einem vollständigen Zeitplan Task T3 (der häufigste, höchste Priorität) seine Deadlines reißen, weil die beiden anderen zusammen zu viel CPU-Zeit verbrauchen. Unter Annahme von RMS-Prioritäten (T3 > T1 > T2) käme es zur Überlast.
> Für **EDF** prüfen wir: Hier ist die notwendige und hinreichende Bedingung $U ≤ 1$. Wir haben $U ≈ 0.938 ≤ 1$, also ist die Bedingung erfüllt [73]. EDF *kann* dieses Taskset demnach planen. Im Unterschied zu RMS würde EDF in manchen Frames dem Task T2 (Deadline 16) später Priorität geben als T1 (Deadline 8) – sodass die Deadlines gerade eingehalten werden. Insgesamt lässt EDF eine höhere Auslastung zu als RMS, in diesem Beispiel etwa 93.8%.

Synchronisationsprobleme und -lösungen (Priority Inversion, Bitstuffing, CAN, ARINC)

Zum Abschluss betrachten wir einige typische Synchronisations- und Kommunikationsprobleme in Echtzeitsystemen sowie Mechanismen zu deren Beherrschung. Diese umfassen sowohl Probleme der **Task-Synchronisation** (z.B. Prioritätsumkehr) als auch der **Kommunikationstiming** (z.B. Jitter durch Bitstuffing).

Priority Inversion (Prioritätsumkehr): Dieses Phänomen tritt auf, wenn in einem Prioritätensystem ein hochpriorisierter Task indirekt von einem niedriger priorisierten blockiert wird [74]. Ein klassisches Szenario: Task H (hoch) und Task L (low) nutzen eine gemeinsame Ressource R (geschützt durch Mutex). Task L läuft und hält gerade R. Nun wird Task H bereit – eigentlich sollte H sofort laufen (höhere Priorität), **kann aber nicht**, weil R noch von L besetzt ist. H wird also blockiert, *wartet* auf R. Inzwischen läuft eventuell ein Task M (mittel) weiter, der weder R braucht noch so hoch priorisiert ist wie H, aber höher als L. Dieser Task M **verdrängt** Task L von der CPU, obwohl L dringend R freigeben müsste für H. Somit steht H hinter M – die niedrige Priorität L hat *effektiv* die hohe H ausgebremst, indem sie von M unterbrochen wurde. Die hohe Priorität "invertiert" sich also gewissermaßen nach unten. Dieses Problem kann unbeschränkt andauern, wenn immer neue M-Tasks kommen, während H wartet. Lösung dafür bieten **Protokolle zur Prioritäten-Vererbung**. Beim *Priority-Inheritance Protocol* gilt: Wenn ein **Low-Priority-Task L einen High-Priority-Task H blockiert**, indem er eine benötigte Ressource hält, **erbt L vorübergehend die hohe Priorität** von H [75]. Dadurch kann kein Task der mittleren Priorität L mehr unterbrechen – L läuft mit hochgeerbter Priorität weiter, bis er die Ressource freigibt [76]. Danach fallen alle Prioritäten wieder auf die ursprünglichen Werte zurück. Priority Inheritance stellt sicher, dass

ein Low-Task die wichtigen High-Tasks nicht unnötig lange aufhält – L "arbeitet schnell fertig". Dieses Protokoll **verhindert jedoch keine Deadlocks** (gegenseitiges zyklisches Warten bleibt möglich) und es kann zu **verzögerter Freigabe** kommen, wenn L mit geerbter Priorität noch länger als nötig läuft. Deshalb gibt es strengere Verfahren.

Das **Priority Ceiling Protocol (PCP)** verbessert die Situation, indem es jeder Ressource R im System einen festen **Prioritätsgrenzwert (Ceiling)** zuordnet – nämlich gleich der statischen Priorität des höchstpriorisierten Tasks, der R *benutzen darf* [77]. Beispiel: Ressource R1 wird von Task H und M benutzt (H > M), also Ceiling(R1) = Priorität(H). Die Regeln beim PCP lauten vereinfacht: Sobald ein Task eine Ressource lockt, wird seine **effektive Ausführungspriorität auf das Level des Ceiling dieser Ressource angehoben** [78]. Er "läuft also so hoch, wie nötig", solange er diese Ressource hält. Weiterhin darf ein Task **eine Ressource nur anfordern, wenn seine Priorität höher ist als die Ceilings aller aktuell von anderen Tasks gehaltenen Ressourcen** [79]. Damit wird verhindert, dass ein Task in eine Situation kommt, wo er von jemand Niedrigerem blockiert werden könnte, während ein Dritter ihn verdrängt – die klassische Dreiecks-Inversion wird ausgeschlossen. PCP sorgt dafür, dass **Priority Inversion maximal einmal pro Ressource** auftritt und begrenzt ist (genauer: höchstens um die Dauer eines kritischen Abschnitts eines Low-Tasks pro höherem Task). Außerdem kann man zeigen, dass bei strikt befolgtem Priority Ceiling **Deadlocks unmöglich** sind, da zyklisches Warten durch die Ceiling-Regel unterbunden wird – ein Task kann sich nur dann eine weitere Resource nehmen, wenn er niemanden mehr damit blockieren würde. Priority Inheritance und Priority Ceiling sind in vielen Echtzeitbetriebssystemen implementiert (z.B. als Option in POSIX Echtzeit-Mutexen). Wichtig ist, diese Mechanismen gezielt für gemeinsam genutzte Ressourcen einzusetzen, um die *Blocking Times* vorhersehbar zu machen.

Bitstuffing und Jitter: Im Bereich der Kommunikation taucht das Problem des **Datenabhängigen Jitters** auf. Ein prominentes Beispiel ist das **Bitstuffing**. Viele serielle Protokolle (z.B. CAN) verwenden eine codierte Übertragung, bei der die Takterholung am Empfänger durch regelmäßige Signalflanken sichergestellt werden muss. Bitstuffing bedeutet, dass bei einer langen Folge identischer Bits automatisch ein **gegenteiliges Bit eingefügt** wird, um einen Pegelwechsel zu erzeugen [80]. Konkret im CAN-Protokoll: Nach fünf aufeinanderfolgenden `0`-Bits fügt der Sender automatisch ein `1`-Bit ein (und umgekehrt würde man nach fünf `1`en eine `0` einfügen, wobei im CAN nominell niemals fünf 1en in Folge ohne Stuff-Bit auftreten, da `1` auch als Delimiter benutzt wird) [80]. Dieses extra **Stuff-Bit** trägt keine Information außer Synchronisation. Der Empfänger entfernt es beim Lesen. *Warum ist das nötig?* – Ohne Bitstuffing könnte ein Signal beispielsweise lange auf `Low` bleiben; der Empfänger läuft mit einem eigenen Takt weiter und würde, ohne Flanken, irgendwann auseinanderlaufen (mehr oder weniger Bits lesen als gesendet wurden) [81]. Durch das erzwungene Inversionsbit nach einer gewissen Länge (bei NRZ-Codierung typisch nach 5 Bits) wird gewährleistet, dass Sender und Empfänger-Uhr nicht zu stark auseinander driften [81]. **Nachteil:** Bitstuffing verändert die effektive Länge des Datenframes je nach Inhalt. Worst-Case müssen alle 5 Bits ein Stuff-Bit eingefügt werden, was z.B. bei einem langen CAN-Datenfeld aus lauter Nullbits die maximale Frame-Dauer erhöht. Dies führt zu geringfügigem **Jitter** in der Übertragungszeit – die Latenz einer Nachricht hängt nicht nur von fester Größe ab, sondern auch von ihrem konkreten Bitmuster. In Echtzeitsystemen muss dieser Jitter in der Timing-Analyse berücksichtigt werden (z.B. nimmt man den Worst-Case mit maximalen Stuffbits an).

CAN-Bus und deterministische Arbitration: Der **Controller Area Network (CAN)** Bus wurde schon bei den Kommunikationsmechanismen angesprochen. Hier fassen wir die relevanten Synchronisationsaspekte zusammen: CAN nutzt ein **Multi-Master-Bus** mit kollisionsfreiem *bit-arbitration* Verfahren (CSMA/CA). Durch die feste Priorisierung über die Message Identifier kann es zwar zu Verzögerungen niedriger Prioritäten kommen, aber keine zerstörerischen Kollisionen – das Protokoll löst den Zugriff deterministisch nach Priorität. Probleme wie **Priority Inversion auf Bus-Ebene** sind dadurch mitigiert, weil ein wichtiger Knoten sich auf Buslevel nicht von unwichtigeren verdrängen lässt

(solange er senden *will*, gewinnt er gegenüber allen gerade ebenfalls sendewilligen Niedrigeren). Allerdings kann es in extremer Last passieren, dass niedrige Prioritäten sehr lange verhungern (Starvation), was in der Analyse zu beachten ist. Ein weiterer Aspekt ist die **Fehlerbehandlung** im CAN: Durch Fehlersignalisierung (Error Frames) und Retransmission kann zusätzlicher Jitter entstehen – im schlimmsten Fall können bestimmte Bitfehlermuster ebenfalls die Buslast erhöhen. Dennoch gilt CAN als *echtzeitfähig* für viele Anwendungen bis etwa 40–50% Buslast, weil dann die Verzögerungen begrenzt und berechenbar bleiben. Wichtige Parameter, die die Synchronisation beeinflussen, sind hier die **Ausbreitungsverzögerung** auf dem Bus (Signal-Laufzeit) und die **Sampling Points** der Empfänger, die eng toleriert synchronisiert sein müssen (das wiederum bestimmt den Bit Time und damit die Baudrate). Die Taktquarze der CAN-Knoten laufen eigenständig und werden durch eine *Resynchronisation* an dominanten Flanken immer wieder justiert – diese Mechanismen zusammen mit Bitstuffing stellen sicher, dass der gesamte Bus weitgehend synchron bleibt und alle Knoten innerhalb eines Bit-Zeitrasters die Bits lesen. (In hochdeterministischen Anwendungen wird CAN allerdings an seine Grenzen stoßen, weshalb dort zeitgesteuerte Protokolle oder CAN-Zeittrigger-Erweiterungen genutzt werden.)

ARINC 629 (Minislotting): Als Kontrast zu CANs Event-Bus sei **ARINC 629** (z.B. genutzt in Boeing-Flugzeugen) erwähnt. Dieses Protokoll implementiert ein **verteilt geregeltes TDMA-ähnliches Verfahren** mittels *Minislots*. Jeder Knoten hat hier eine feste Wartezeit, bevor er senden darf, anstatt dass alle sofort versuchen. Im Prinzip gibt es bei ARINC 629 drei wichtige Zeitparameter [82]:
- Ein **Timing Interval** (TAF, Terminal Access Frame), das für *alle* Knoten gleich ist und sicherstellt, dass niemand den Bus monopolisiert – es begrenzt die maximale Sendelänge bzw. definiert einen Zyklus, nach dem alle anderen mal dran gewesen sein müssen [82].
- Einen **Terminal Gap (TG)** pro Knoten, der unterschiedlich lang ist (in Anzahl *Minislots*) und bestimmt, wann ein Knoten nach Busfreigabe zu senden beginnen darf [83]. Der TG ist stets größer als die maximale Signallaufzeit auf dem Bus, so dass keine zwei Knoten exakt gleichzeitig anfangen können – der mit dem kürzesten TG gewinnt den Zugriff [84] [85].
- Einen **Synchronization Gap (SG)**, der nach jeder Sendeperiode eingefügt wird und länger ist als der längste TG [85]. In diesem Synchronisationsintervall hat kein Knoten Sendezulass, es dient als “Reset” des Wartesaals: alle Knoten wissen, wenn der SG vorbei ist, beginnt eine neue Wettbewerbsrunde, und jeder wartet wieder seinen TG ab [82].

Stark vereinfacht kann man sich ARINC 629 wie einen zweistufigen Wartesaal vorstellen [86]: Zuerst landen alle sendewilligen Knoten in einem virtuellen Warteraum (während der TG-Phase), und der mit dem kleinsten TG verlässt den Raum zuerst und sendet seine Nachricht. Dann dürfen im *gleichen Zyklus* auch die anderen wartenden Nodes nacheinander senden (in festgelegter Reihenfolge oder nach erneutem Minislot-Check), bevor neue in den Warteraum kommen [86]. Das Timing Interval begrenzt dabei die Zyklusdauer für einen Knoten, damit keiner dauerhaft den Bus blockiert. Dieses verteilte Token-like System benötigt keine zentrale Arbiterinstanz und funktioniert deterministisch, solange alle Knoten sich an die voreingestellten Timer halten. Für Echtzeitsysteme hat ARINC 629 den Vorteil, *Kollisionen völlig zu vermeiden* und Bandbrete fair zu teilen – allerdings auf Kosten von etwas mehr Durchsatzlatenz und Komplexität bei der Parametrierung (die TGs müssen klug gewählt werden, damit Prioritäten abgebildet werden können). Moderne zeitgesteuerte Busse (wie ARINC 664, AFDX – ein deterministisches Ethernet) lösen ähnliche Probleme mittels virtueller Links und Bandbreitenbegrenzungen. Wichtig bleibt: Egal ob CAN, ARINC 629 oder andere – die Synchronisationsmechanismen auf dem Bus müssen in der Worst-Case-Analyse berücksichtigt werden.

> **Beispielaufgabe:** *„Was versteht man unter Bitstuffing und wofür wird es benötigt?“*
>
> **Lösung: Bitstuffing** bezeichnet das automatische Einfügen eines invertierten Bits in einen Datenstrom nach einer bestimmten Anzahl gleichbleibender Bits [80]. Bei einer

Non-Return-to-Zero Kodierung (wie im CAN-Bus) bedeutet das konkret: Nach fünf identischen Bits fügt der Sender ein entgegengesetztes Bit ein [80]. Dieses zusätzliche Bit enthält keine Nutzinformation, sondern dient rein der **Taktrückgewinnung** beim Empfänger. Ohne Bitstuffing könnten Sender und Empfänger während langer Phasen ohne Pegelwechsel auseinanderlaufen – d.h. der Empfänger würde womöglich mehr oder weniger Bits empfangen als gesendet wurden [81]. Durch die eingefügten Bits wird sichergestellt, dass spätestens nach 5 Bits eine Flanke auftritt, an der der Empfänger seinen Takt synchronisieren kann. Bitstuffing verhindert also, dass die Empfängeruhr zu stark vom Sender abweicht, **erkauft** dies jedoch mit einem leichten Overhead und **Jitter**: Die effektive Frame-Länge kann variieren, weil abhängig vom Dateninhalt mehrere Stuff-Bits eingefügt werden können [87]. In Echtzeit-Bussystemen wie CAN wird dieser Jitter bei der Laufzeitberechnung berücksichtigt, indem man im Worst-Case annimmt, dass das ungünstigste Bitmuster (maximale Stuffbits) vorliegt. Damit bleibt trotz Bitstuffing die Einhaltbarkeit von Deadlinezeiten belegbar.

---

1 2 3 4 6 7 9 10 11 12 13 14 rts02_time_and_order.pdf
file://file-Hp2zXBnh2i7qREB7Y189LF

5 39 40 80 81 87 Echtzeitsysteme Klausur.pdf
file://file-9ejZpPp1Ac84VkN6tYLLTu

8 15 55 56 57 58 59 60 70 71 72 73 75 76 77 78 79 82 86 Fragenkatalog ES.pdf
file://file-NTwi3n6Eo6ZkJ7ESyc2MTi

16 17 18 23 30 31 32 33 47 48 49 50 51 52 ES_Fragenkatalog_71.pdf
file://file-DeB4hcNG4DpxYa1CvjGvjM

19 20 21 22 24 25 26 rts03_rt_model.pdf
file://file-DNRiJGLySjnTGE4MEPiy29

27 28 29 83 84 85 rts04_rt_communication.pdf
file://file-5MV7Tja3ghFQGWAGiMnoXA

34 35 36 37 38 41 42 43 44 45 46 53 54 rts05_component_services.pdf
file://file-Q9PpkgdCWQ97DaizpJ8ivy

61 62 63 64 65 66 67 68 69 74 rts07_rt_scheduling.pdf
file://file-8eFp2oRynxYnQSLD8oe3fa