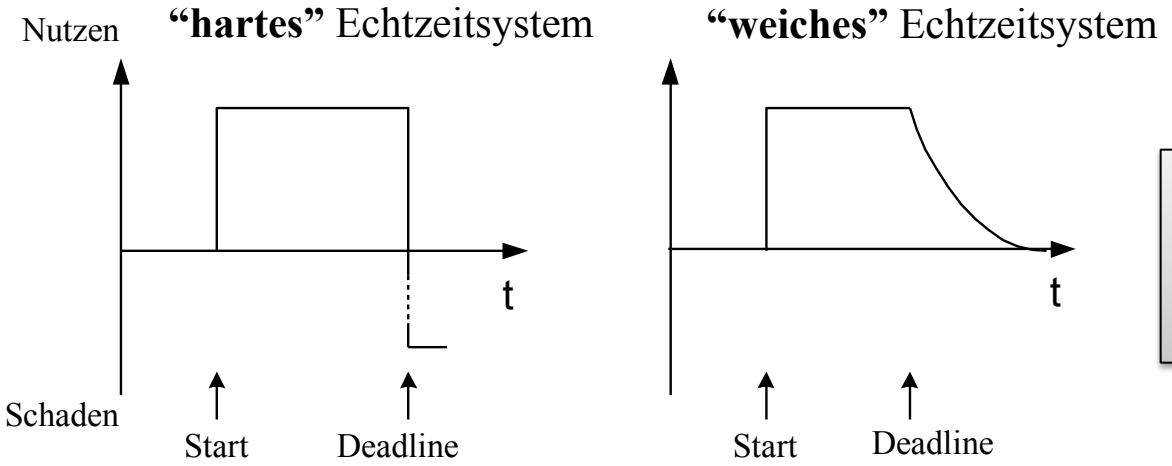
Kapitel 2: Echtzeitsysteme

Was ist ein Echtzeitsystem?

- Echtzeitsysteme sind technische Systeme, die korrekte Reaktionen innerhalb eines definierten Zeitlimits produzieren müssen.
- Wenn die Reaktionen diese Zeitlimits überschreiten, dann resultieren daraus Leistungseinbußen oder Fehlfunktionen.
- Ein logisch korrektes Ergebnis zum falschen Zeitpunkt kann die Sicherheit eines Systems gefährden.
- „Echtzeit“ oder „Realzeit“ bedeutet, den echten bzw. realen Zeitabläufen entsprechend. Eine Zeitdehnung oder Raffung ist nicht zulässig.
- Man unterscheidet harte und weiche Echtzeitsysteme.



Echtzeit oder synomym Realzeit (engl. real time) wird häufig mit RT abgekürzt:
RTS = Real Time System
RTOS = Real Time Operating System

Harte vs. weiche Echtzeit

- **Weiche Echtzeitsysteme**
  Es gibt zeitliche Fristen bis zu denen ein Ergebnis berechnet werden muss. Werden diese Fristen nicht eingehalten, hat dies negativen Einfluss auf die Qualität bzw. die Dienstgüte. Es gibt einen „weichen“ Übergang bis zum Totalausfall.
  Beispiel: Video- oder Audio-Übertragung
  Mögliche Folgen: Bild ruckelt, Artefakte entstehen.

- **Harte Echtzeitsysteme**
  Kann die Berechnungsfrist nicht eingehalten werden, hat dies möglicherweise katastrophale Folgen. Die Grenze ist „hart“.
  Beispiel: Raketen- oder Flugzeugsteuerung
  Mögliche Folgen: Absturz und Tod von Menschen.

Definition von Echtzeit nach DIN 44300 [1985]

„Echtzeitbetrieb ist ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig derart betriebsbereit sind, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.“

Missverständnisse bei Echtzeitsystemen

- Echtzeitsysteme sind nicht zwangsläufig besonders “schnelle” Systeme. Entscheidend ist die garantierte Einhaltung der Zeitschranken und die sehr hohe Systemzuverlässigkeit. Eingebettete Echtzeitsysteme sind häufig sogar auf Basis vergleichsweise leistungsschwacher Mikrocomputer realisiert.
- Schnellere Computerhardware ist keine Lösung des Echtzeitproblems. Nicht vorhersehbare Verzögerungen in der Ausführung haben Ihre Ursachen in nicht-deterministischen Ausführungskonzepten (z.B. nonpreemptives Scheduling, zufallsbasierter Medienzugriff bei Ethernet) oder mangelnde Synchronisation von Uhren.
- Echtzeitsysteme werden heute in der Regel nicht mehr in Assembler programmiert. Der größte Teil wird in Hochsprachen, vor allem C, oder den SPS-Fachsprachen programmiert.
- Hundertprozentige Garantien, z.B. zur Einhaltung von Reaktionszeiten, kann es wegen möglicher Hardwarefehler und nie vollständig vermeidbarer Programmierfehler grundsätzlich nicht geben! Bei der Entwicklung von Echtzeitsystemen muss deshalb zusätzlich, beispielsweise durch redundante Auslegung der Software und Hardware, eine möglichst geringe Fehlerwahrscheinlichkeit erreicht werden. Echtzeitsysteme benötigen deshalb die Angabe der geforderten und nachweisbaren Fehlerwahrscheinlichkeit.

Realisierungsformen von Echtzeitsystemen

- Ein Echtzeitsystem kann in Hardware (z.B. FPGA oder Hydraulische Steuerung) sowie mit geeigneter Software auf einem Digitalrechner (z.B. Mikrocontroller, SPS) realisiert werden.
- Computer, die zur Automatisierung (Steuerung und Regelung) technischer Einrichtungen (Maschinen, Anlagen, Fahrzeuge usw.) eingesetzt werden, sind praktisch immer Echtzeitsysteme. Die zeitlichen Anforderungen ergeben sich aus dem Ein- und Ausgabeverhalten eines „echten“ technischen Prozess und der Automatisierungsaufgabe und können sehr unterschiedlich sein (< 1 ms bis mehrere Stunden).
- Beispiele für Echtzeitsysteme:
  - Harte Echtzeitsysteme:
    - Airbag, Anti-Blockier-System (ABS), ESP, ACC etc. (Mikrocontroller)
    - Industrielle Fertigungssteuerung (Speicherprogrammierbare Steuerung)
    - Heizungsregler (ASIC/FPGA oder Mikrocontroller)
    - Motorsteuerung (Relaissteuerung)
    - Digitaler Messdatenlogger (Digitaler Signalprozessor)
    - Prozessleitsteuerung (Prozessrechner)
    - Maschinensteuerung (CNC)
  - Weiche Echtzeitsysteme:
    - TV-Sportübertragung
    - Navigationssystem im Auto
    - Telefonanlage

Eingebettetes System

- Technisches System, das durch ein integriertes, programmiertes Computersystem auf Basis eines Mikroprozessor gesteuert wird.
- Das Computersystem ist nach außen in der Regel nicht direkt sichtbar. Man spricht dann von einem eingebetteten Computersystem.
- Das Computersystem ist über spezielle Schnittstellen in Form von Sensoren und Aktoren mit dem technischen System vielfältig verbunden.
- Häufig werden leistungsärmere, energieeffiziente aber robuste Mikroprozessoren mit Einschränkungen in Bezug auf Rechenleistung und Speicherfähigkeit eingesetzt.
- Praktisch immer werden Anforderung zur harten oder wenigstens weichen Echtzeitfähigkeit gestellt.
- Mehrere verbundene, eingebettete Systeme bilden ein verteiltes System und damit auch ein verteiltes Echtzeitsystem.

Anforderungen an ein Echtzeitsystem

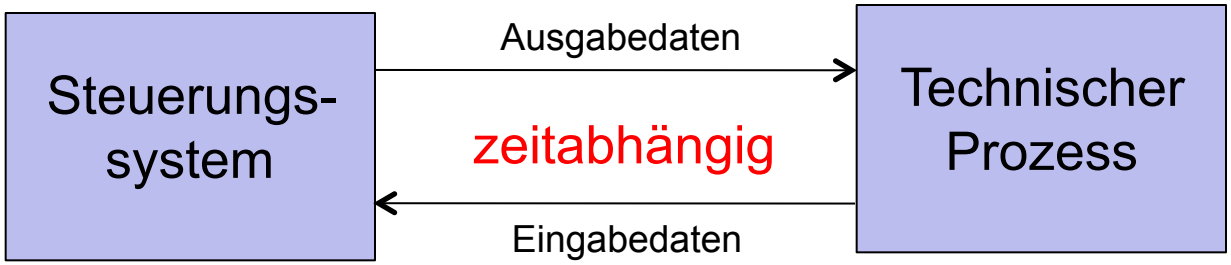
Zentrale Anforderungen sind:

- Rechtzeitigkeit
- Gleichzeitigkeit

Weitere, wichtige Anforderungen sind außerdem:

- Sicherheit (keine Gefährdung von Menschen und Sachen)
- Verfügbarkeit / Zuverlässigkeit
- Robustheit / Adaptivität
- Unterbrechbarkeit
- Korrektheit / Genauigkeit
- Existenz von Datenschnittstellen zum externen Prozess

Forderung nach Rechtzeitigkeit



- Eingabedaten müssen rechtzeitig abgerufen werden (Kriterien sind Dauer und Frequenz von Signalen und das Abtasttheorem).
- Ausgabedaten müssen rechtzeitig verfügbar sein (max. Reaktionszeit, Regelgüte und Stabilität).
- Zu unterscheiden sind:
  - Absolutzeitbedingungen (bezogen auf eine genaue, globale Uhrzeit T)
  - Relativzeitbedingungen (zeitlich relativ zu bevorstehenden Ereignissen $T_E \pm \delta t$)
  - Feste, äquidistante Zeitpunkte (genau zu diesem Zeitpunkt T bzw. k*T)
  - früheste Zeitpunkte (nicht bevor T oder $T_E - \delta t$)
  - späteste Zeitpunkte (nicht nach T oder $T_E + \delta t$)
  - Toleranzintervall (nicht bevor T_1 und nicht nach T_2 oder zwischen $T_E - \delta t$ und $T_E + \delta t$)

Forderung nach Gleichzeitigkeit

- Die Forderung nach Gleichzeitigkeit ergibt sich aus der Tatsache, dass Echtzeitsysteme auf Vorgänge in ihrer Umgebung reagieren müssen, die dort gleichzeitig ablaufen (unabhängige Zustandsgrößen).
- Lösbar dadurch, dass für jeden zeitlich parallelen Vorgang (Zustandsgröße) ein eigener Computer oder Steuerungssystem eingesetzt wird.
- Die große Rechenleistung heutiger Prozessoren erlaubt aber die quasi gleichzeitige Ausführung mehrerer unabhängiger Programme. Jedes Programm wird dazu beispielsweise von einem entsprechenden Echtzeitbetriebssystem immer nur für eine kurze, begrenzte Zeit ausgeführt, bevor es erneut unterbrochen wird, und das nächste Programm gestartet wird. Sind alle Programme an der Reihe gewesen, beginnt das Ganze zyklisch wieder von vorne (Round Robin). Solche Programme werden meist Task genannt.

Kapitel 2: Echtzeitsysteme

2.1 Grundbegriffe

2.2 Programmierung von Echtzeitsystemen

2.3 Organisation von Echtzeitsystemen

Task

Definition: Der Task ist der zeitliche Vorgang der Abarbeitung eines sequentiellen Programms auf einem Computer. Ein Task existiert nicht nur während der Ausführung der Befehle, sondern auch während geplanter oder erzwungener Wartezeiten. Er beginnt mit dem Eintrag in eine Liste des Echtzeitbetriebssystems und endet mit dem Löschen aus dieser Liste. Tasks können unterschiedliche Zustände einnehmen, wie „geplant“, „aktiviert“, „angehalten“, „bereit“, „ruhend“ u.ä.

Der Begriff „Task“ wird nicht einheitlich verwendet. Er wird auch als Oberbegriff für „Thread“ und „(Rechen-)Prozess“ oder synonym dazu verwendet.

Multitasking bezeichnet dann den Vorgang mehrere Tasks quasi gleichzeitig auf einem Computer auszuführen. Die Organisation der Taskausführung erfolgt durch den Scheduler, der Teil eines Echtzeitbetriebssystems ist. Die Ausführung des Task erfolgt zeit- und prioriätengesteuert. Können laufende Tasks durch das Betriebssystem unterbrochen werden, spricht man von präemptivem Multitasking.

Programmierung von Echtzeitsystemen

- Planung des zeitlichen Ablaufs der Tasks vor ihrer Ausführung (Zeitsteuerung)
  ➔ synchrone Programmierung
  - Der gesamte zeitliche Systemablauf wird zur Übersetzungszeit festgelegt.
  - Voraussetzung ist eine präzise, globale Uhr insbesondere bei verteilten Systemen oder eine feste zyklische Ausführung mit festen Codepfaden.
  - Je Task wird ein festes Zeitfenster reserviert. Dazu ist die Abschätzung der maximalen Laufzeit (WCET) notwendig.
- Organisation des zeitlichen Ablaufs während der Ausführung der Programme (Ereignissteuerung)
  ➔ asynchrone Programmierung
  - Ereignisse (Interrupts) triggern die zugehörigen Funktionen.
  - Der Aufruf der Tasks erfolgt dynamisch zur Laufzeit (Scheduling). Prioritäten regeln mögliche Konflikte und die Dringlichkeit.
  - Mögliche Unterbrechungen und Verschiebungen durch höher priorisierte Tasks müssen bei der garantierten Antwortzeit berücksichtigt werden.

Synchrone Programmierung

- Die Tasks werden nach einem festen Zeitplan gestartet. Man nennt dies auch Zeitsteuerung.
- Jedem Task wird eine genau definierte Zeitscheibe zugeteilt, in der er seine Berechnungen abschließen muss.
- Eine Überlastung des Echtzeitsystems kann dann ausgeschlossen werden, wenn die vorab bestimmten Laufzeiten die zugeteilten Zeiten sicher nicht überschreiten. Die Laufzeiten müssen dazu verlässlich abgeschätzt bzw. berechnet werden können.
- Die synchrone Programmierung erfordert einen hohen Planungsaufwand vor bzw. während der Implementierung. Jede spätere Änderung muss erneut hinsichtlich Ihrer Echtzeitfähigkeit überprüft werden, gegebenenfalls muss umgeplant werden.
- Wenn die benötigten Ausführungszeiten exakt bekannt sind kann der Prozessor zu nahezu 100% ausgelastet werden.
- Wenn die Tasks jeweils einzeln durch exakte Uhren getriggert werden, kann die zeitliche Unschärfe (Jitter), die sich beispielsweise aus nicht kontrollierbaren Einflüsse aus der Computerhardware oder dem Betriebssystem ergeben, minimiert werden.

Beispiel für synchrone Programmierung

Pseudocode für Assembler auf Intel-Mikrocontroller

```
 mov a, 10000 ; Grenzwert der Drehzahl
 mov b, 30    ; Grenzwert der Temperatur
 mov O, 1     ; Abschaltsignal

:loop         ; Markierung im Programmfluss (keine Instruktion)
 in t,PORT1   ; einlesen der aktuellen drehzahl-Werte
 in d,PORT2   ; einlesen der aktuellen temp-Werte

:drehcheck
 cmp t,a      ; prüfe die Drehzahl
 jg  tempcheck; wenn Grenzwert nicht erreicht, springe zu :tempcheck
 out PORT3,O  ; Grenzwert erreicht! setze Abschaltsignal

:tempcheck
 cmp d,b      ; prüfe die Temperatur
 jg  loop     ; wenn Grenzwert nicht erreicht, springe zu :loop
 out PORT3,O  ; Grenzwert erreicht! setze Abschaltsignal

 jmp loop     ; unbedingter Sprung zur Marke :loop (Endlosschleife)
```

zeitlich synchrone Erfassung der Sensorwerte

Quelle: Programmcode siehe Wikipedia “Echtzeitsysteme”

Endlos-Programmschleife (ohne Echtzeitbetriebssystem und Tasks) mit der auf das Überschreiten einer Temperatur t oder einer Drehzahl d durch Abschalten reagiert werden soll. Die maximale Reaktionszeit bis zur Ausgabe des Abschaltsignals, kann exakt berechnet werden, da die Ausführungszeit jedes einzelnen Befehls bekannt ist. Sie kann hier grob mit der zweifachen Zykluszeit der Endlosschleife nach oben abgeschätzt werden (WCET << 1ms). Die exakte Reaktionszeit ist im Einzelfall aber nicht vorab bestimmbar, da dies davon abhängt, zu welchem Zeitpunkt relativ zum zeitlichen Status der Ausführung der Endlosschleife, die Abbruchbedingung erstmals erfüllt wird.

Asynchrone Programmierung

- Ziel ist es, auf ein von außen kommendes Ereignis (z.B. Interrupt) schnellstmöglich zu reagieren. Dies nennt man auch Ereignissteuerung.
- Gegebenenfalls wird dazu ein gerade laufender Task unterbrochen.
- Auf ein einzelnes Ereignis kann so mit sehr geringem Zeitverlust reagiert werden.
- Wenn mehrere Ereignisse zeitgleich oder sehr zeitnah auftreten, kommt es zum Konflikt. Hierfür muss es eine Strategie geben, diesen Konflikt deterministisch aufzulösen (Steuerung über Prioritäten).
- Zur Abschätzung bzw. Berechnung der maximalen Reaktionszeit, muss bekannt sein, wie häufig die einzelnen Ereignisse auftreten können und wie viele andere Tasks jeweils höher priorisiert sind und wie lange diese zur vollständigen Ausführung brauchen.
- In einer harten Echtzeitumgebung muss sichergestellt werden, dass selbst der Task mit der niedrigsten Priorität immer noch rechtzeitig sein Ergebnis abliefern kann (Puffer). Die Rechenleistung kann daher bei der asynchronen Programmierung nie zu 100% genutzt werden.

```
/* Initialisierungsfunktion beim Treiberstart */
void rtcan_register(struct rtcan_device *dev) {
    /* RTDM (Real-Time Driver Model) ist das Echtzeit-Treibermodell von Xenomai.
     * dev: Gerätestruktur für Verwaltungsinformationen
     * irq_handle: IRQ-spezifische Verwaltungsinformationen
     * rtcan_interrupt_handler: Interrupt-Handler-Routine */
    rtdm_irq_request(&dev->irq_handle, irq_number, rtcan_interrupt_handler, irq_flags, name, dev);
}

/* Treiber beenden */
void rtcan_unregister(struct rtcan_device *dev) {
    rtdm_irq_free(&dev->irq_handle);
}

/* Interrupt-Handler, wird in rtcan_register() registriert (siehe oben). */
int rtcan_interrupt_handler(rtdm_irq_t *irq_handle) {
    /* Zuerst muss festgestellt werden, welche Art von Interrupt aufgetreten ist.
     * Dazu wird ein CAN-Hardware-spezifisches Interrupt-Register (IR) ausgelesen. */
    u8 irq_source = read_reg(dev, IR);

    /* Transmit-Interrupt? */
    if (irq_source & IR_TI)
        /* Sender signalisieren, dass der Transmit einer CAN-Nachricht abgeschlossen ist
                * und der Programmfluss fortgeführt werden kann. */
        rtdm_sem_up(&dev->tx_sem);
    /* Receive Interrupt? */
    else if (irq_source & IR_RI) {
        /* CAN-Nachricht aus Hardware-Registern auslesen. Dies geschieht in einer
                * separaten Funktion (hier nicht angegeben). */
        rtcan_rx_interrupt(dev, &can_frame_buffer);
                /* überreiche die empfangene CAN-Nachricht an die interessierten Tasks. */
        rtcan_rcv(dev, &can_frame_buffer)
    }
    /* Bestätige der CAN-Hardware die Beendigung der Interrupt-Abarbeitung. */
    irq_ack(dev);

    return RTDM_IRQ_HANDLED;
}
```

A) siehe nächste Folie

Dieser Code ist unvollständig und soll nur das Grundprinzip verdeutlichen.

Beispiel für asynchrone Programmierung

Ausschnitt aus C-Systemfunktion in Xenomai (CAN-Treiber)

Grundprinzip: Zunächst muss für die spätere Reaktion auf einen möglichen Interrupt die Adresse des Interrupthandlers (Task) in einer Tabelle abgelegt werden. Kommt dann später der Interrupt, wird das laufende Programm gestoppt und der zugehörige Interrupthandler aus dieser Tabelle heraus gestartet. Der Interrupthandler kehrt nach Bearbeitung zum Betriebssystem zurück, das dafür sorgt, dass das ursprüngliche Programm fortgesetzt wird.

XENOMAI-Anwendungs-programm

das CAN-Nachrichten in Echtzeit verarbeitet (Ausschnitt)

```
#define DREHZAHL_GRENZWERT      100
#define TEMP_GRENZWERT          30
#define ABSCHALTSIGNAL          1

#define DREHZAHL_CAN_ID         1
#define TEMP_CAN_ID             2
#define ABSCHALT_CAN_ID         3

struct can_frame recv_frame;
struct can_frame send_frame;

void can_task()
{
   int sock_recv = rt_dev_socket(PF_CAN, SOCK_RAW, CAN_RAW);
   int sock_send = rt_dev_socket(PF_CAN, SOCK_RAW, CAN_RAW);

   send_frame.can_id = ABSCHALT_CAN_ID;
   send_frame.can_dlc = 1;
   send_frame.data[0] = ABSCHALTSIGNAL;

   while (1) {
      rt_dev_recv(sock_recv, &recv_frame, sizeof(struct can_frame), 0);

      if ((recv_frame.can_id == DREHZAHL_CAN_ID) && (recv_frame.data[0] > DREHZAHL_GRENZWERT))
         rt_dev_send(sock_send, &send_frame, sizeof(struct can_frame), 0);

      if ((recv_frame.can_id == TEMP_CAN_ID) && (recv_frame.data[0] > TEMP_GRENZWERT))
         rt_dev_send(sock_send, &send_frame, sizeof(struct can_frame), 0);
   }
}
```

A) siehe vorhergehende Folie

Deterministische Ausführungszeiten

Um Ausführungszeiten im Betrieb garantieren zu können (WCET worst case execution time), dürfen bestimmte Techniken bei Echtzeit-Computersystemen nicht oder nur sehr eingeschränkt verwendet werden, wie z.B.:

- Dynamische Speicherallokation und automatische Garbage Collection.
- Rekursive Funktionsaufrufe, wenn Rekursionstiefe nicht bekannt.
- Warten auf externe Eingaben oder Bestätigungen in Programmschleifen (z.B. Prozesssignale, Benutzereingaben, Speicherzugriffe o.ä.)
- Zufallsgesteuerte Medienzugriffe (z.B. Netzwerk-Kommunikation mit zufälliger Wartezeit nach Kollision)
- Virtueller Speicher (unkalkulierbare Wartezeiten beim Auslagern und Nachladen von Speicherblöcken)
- Betriebssystemaufrufe (nur wenn max. Ausführungsdauer garantiert)
- u.ä.

Kapitel 2: Echtzeitsysteme

2.1 Grundbegriffe

2.2 Programmierung von Echtzeitsystemen

2.3 Organisation von Echtzeitsystemen

Echtzeitbetriebssystem

Nach DIN 44300 versteht man unter einem Betriebssystem die Programme eines Computers, die zusammen mit den Eigenschaften der Computerhardware die Grundlage der möglichen Betriebsarten des Computersystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen.

Ein Echtzeitbetriebsystem (engl. Real-Time Operating System) muss zusätzlich die Bedingungen der Gleichzeitigkeit und Rechtzeitigkeit erfüllen. Es muss deshalb innerhalb bestimmter, garantierter Zeiten auf Signale von außen reagieren und deshalb gewährleisten, dass alle Tasks rechtzeitig ausgeführt werden. Zu einem Echtzeitbetriebsystem gehören deshalb zusätzlich die folgenden Komponenten:

- Ein-/Ausgabsteuerung mit Zugriff auf die Schnittstellen zum technischen Prozess über Sensoren und Aktoren (Treiber)
- Interruptsteuerung
- Taskverwaltung (➔Task-Scheduling)

Beispiele: QNX, VxWorks, Linux/Xenomai, RT-Linux, OSEK-OS, T-Kernel, FreeRTOS uvm.

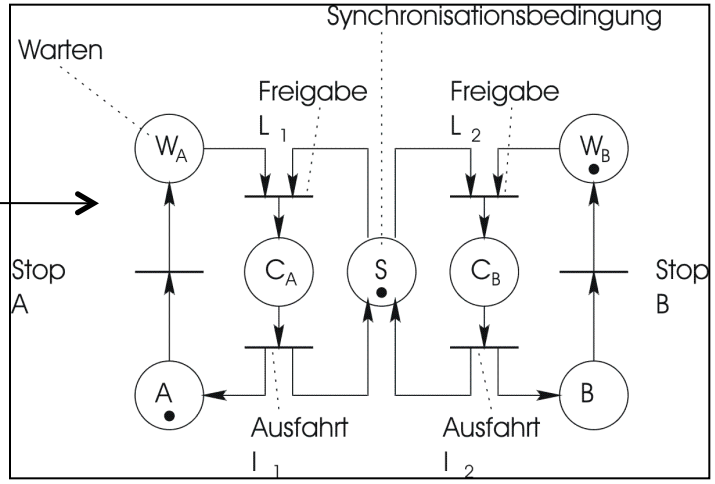
Task-Scheduling

- Ein Programm, das die Zuteilung der Betriebsmittel (CPU, Speicher, E/A usw.) vornimmt, durch das also die Zuteilungsstrategie realisiert wird, wird Scheduler genannt.
- Neben den vom Anwender definierten Tasks muss eine Vielzahl anderer, sogenannter Systemtasks verwaltet werden (Statistiken, Speicherverwaltung, Benutzerinterface usw.).
- Ziel des Schedulers ist es, den effizienten Parallelbetrieb möglichst vieler Betriebsmittel zu erreichen. Dazu gehört z.B., dass wenn ein Task auf eine Eingabe wartet, die CPU sofort für einen anderen Task freigegeben wird. Die gleichzeitige Anforderung eines Betriebsmittels durch mehrere Tasks wird über Warteschlagen verwaltet. Es kommen verschiedene Strategien zur Verwaltung der Warteschlangen zum Einsatz, wie z.B.: EDF (earliest deadline first), FIFO (first in first out), SJN (shortest job next), Round Robin usw.
- Ein besonderer Task ist der Idle-Task (auch Null-Prozess), der immer dann aktiviert wird, wenn kein anderer Task eingeplant und ausführungsbereit ist.

Task-Synchronisierung

- Bei der zeitlich parallelen Ausführung von Tasks kann es vorkommen, dass ein Task einen anderen Task „logisch“ überholt oder dass zwei Tasks gleichzeitig auf das selbe Betriebsmittel zugreifen wollen. Dies muss synchronisiert werden. Man unterscheidet:
  ➔ Logische Synchronisation: Erzwingen von definierten zeitlichen Folgen von Aktionen, d.h. die Synchronisation zwischen den Abläufen im technischen Prozess und den Tasks.
  ➔ Betriebsmittelorientierte Synchronisation: Einhalten von Bedingungen bzgl. der Verwendung gemeinsam benutzter Betriebsmittel (Hardware oder Software).

- Zur Synchronisation solcher Prozessabläufe gibt es verschiedene Verfahren, wie:
  - Semaphore,
  - kritische Regionen,
  - Rendez-vous-Konzept,
  - usw.



Echtzeitkommunikation

In einem verteilten Echtzeitsystem ist es erforderlich, dass auch der Datenaustausch zwischen Tasks oder der Datenaustausch mit dem technischen Prozess in zeitlich deterministischer Weise erfolgt.

Für diesen Zweck wurden Netzwerke mit entsprechenden Protokollen entwickelt, die einen Datenaustausch unter Echtzeitbedingungen ermöglichen. Diese Netzwerke werden Feldbusse genannt. Je nach Nähe zum technischen Prozess unterscheidet man: Sensor/Aktorbus, Feldbus oder Prozessbus.

Kollisionen beim Medienzugriff müssen wegen der sich daraus ergebenden, unkalkulierbaren Verzögerungen verhindert werden. Dies wird erreicht durch Verfahren, wie zyklische Ringbusse, Master/Slave mit Zeitscheiben, Nachrichten mit Prioritäten u.ä.

Beispiele: ASI, Interbus, CAN, Profibus, RTnet, EtherCAT usw.

Echtzeit-Middleware

Unter Middleware versteht man eine Vermittlungssoftware, die den Informationsaustausch zwischen Anwendungen so unterstützt, dass die gesamte Komplexität der Kommunikation zwischen den Programmen für den Programmierer weitgehend verborgen bleibt. Die Programme können auf der gleichen Computerhardware oder auch auf über ein Netzwerkwerk verbundenen aber getrennten Computern ausgeführt werden. Welches Netzwerk mit welchem Protokoll genutzt wird, ist für den Programmierer unsichtbar. Die Kommunikationstechnologie kann entsprechend leicht ausgetauscht werden, ohne dass dies Auswirkungen auf die Anwendungsprogramme hat. Für die Kommunikation kommen abstrakte Konzepte zum Einsatz, wie z.B. Client/Server, Publish/Subscribe oder Mailboxen.

Eine Middleware, die darüber hinaus zeitliche Garantien beim Informationsaustausch gibt, kann als Echtzeit-Middleware bezeichnet werden. Dazu gehört auch die Synchronisation von Uhren im Netzwerk sowie die Verknüpfung absoluter Zeitstempel mit Informationen, wie beispielsweise Messwerten. In einem verteilten Systeme muss die Kommunikation entsprechend über Feldbusse erfolgen.

Beispiele: RT-CORBA, OSA+, OROCOS RTT, SensorNet, RACK usw.

Zusammenfassung

Es wurde eine Einführung in Echtzeitsysteme gegeben. Dabei wurden die folgenden Begriffe überblicksartig vorgestellt und erläutert:

- Echtzeitsystem
- Eingebettetes System
- Harte und weiche Echtzeit
- Rechtzeitigkeit und Gleichzeitigkeit
- Synchrone und asynchrone Programmierung
- Task bzw. (Rechen-)Prozess
- Echtzeitbetriebssystem
- Task-Scheduling
- Task-Synchronisation
- Echtzeitkommunikation
- Echtzeit-Middleware