Real-Time Programming

1494

WS 2021/22

HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Contents

Chapter 1

Real-Time Systems

Digital, computer-based real-time systems became widespread rapidly with the availability of microprocessors getting more powerful and cheaper from year to year.

1.1 Introduction

Real-Time systems, as used in the wide area of automation, have 2 basic requirements,

1. The **logical correctness** of the systems output as a response to its inputs, and
2. The **timeliness** of the outputs available.

For example, this is obvious for an airbag control, for which a correct decision for ignition is required at the right millisecond, otherwise the resulting behaviour can be harmful !

Since there is a ”digital revolution” the past few decades, a strong trend to applications with microprocessors and witch microcontrollers particularly can be observed.

This means, that classical solutions for **real-time systems** (e.g. analog controller circuits) are replaced by **software applications** running on a microprocessor or a microcontroller.

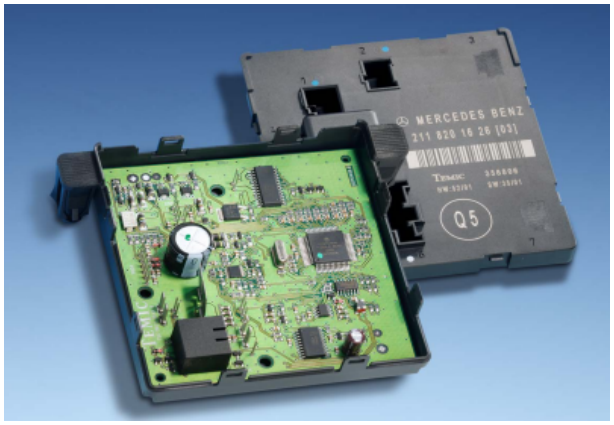
Thus, an important part of **a real-time system is software**. This type of software is much different from commonly used software known from PC applications like office programs, or internet browsers, since real-time requirements have to be met.

In the area of real-time software development, the classical C programming language is widely used, due to its outstanding maturity and flexibility, especially with industrial PC applications, or with microcontroller applications.

Furthermore there is a need for PLC controls (de: SPS) in industrial automation.

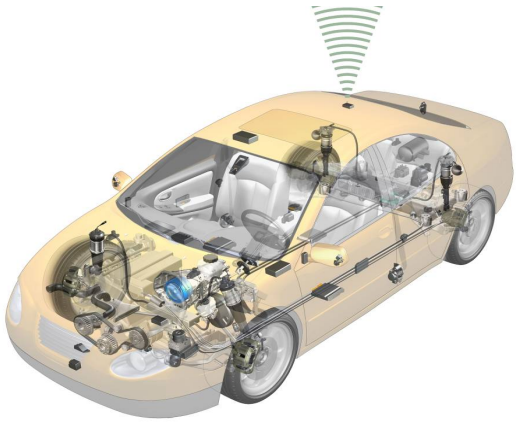
Examples for Real-Time Systems

- Examples of the use of real-time systems are found in the following areas:
    1. Production
    2. Aerospace
    3. Automotive Electronics



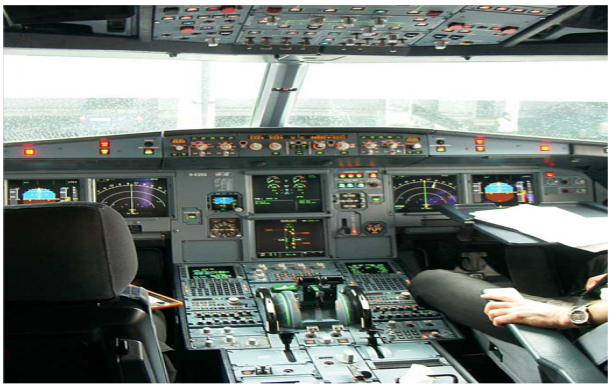
(a) Automotive Door Contol Unit with Anti-Squeeze

(b) Undistorted image

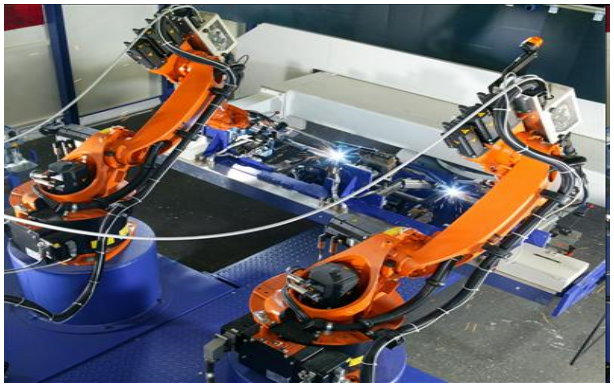


(c) Car with > 100 ECUs

(d) Pacemaker



(e) A319 Cockpit



(f) wikipedia.de: Arc welding of metal parts with industrial robots (KUKA). IPC, PLC

Figure 1.1.1: Comparison of results of video 1 (checkerboard000)

4. Medicine
5. Military Technology,
6. e-Banking,
7. e-Trading,
8. Telecommunications,
9. Network Management,
10. Power Generation/Management,
11. Navigation

The need for Software**(mainly C-Code)in all areas of application grows rapidly**.

Moore's Law: "die Anzahl der Transistoren pro Chipfläche verdoppelt sich alle 2 Jahre"

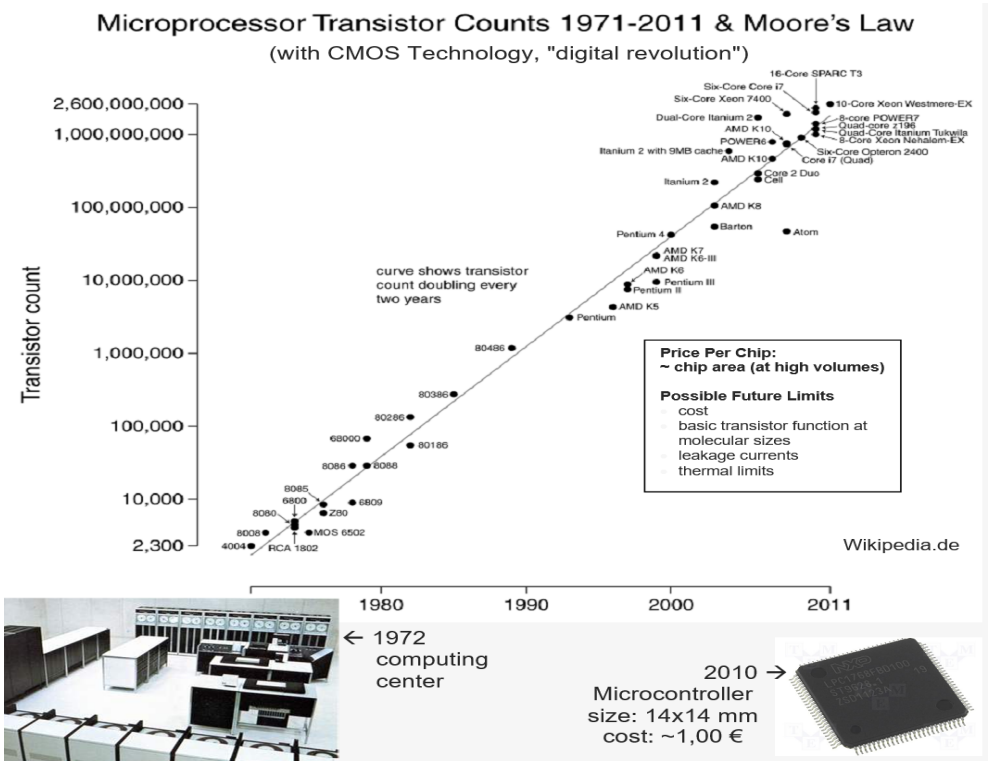


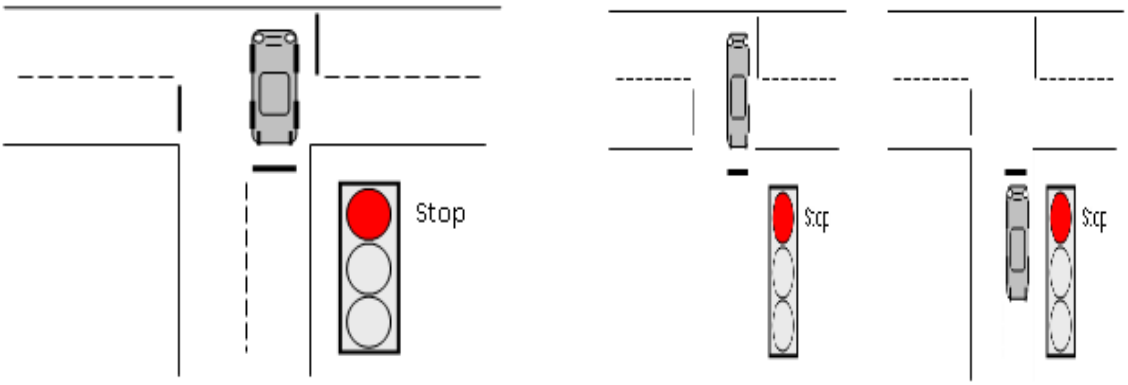
Figure 1.1.2: Moore's Law

Software for Real-Time Systems differs substantially from software for generalpurpose computers (PC) due to the **requirements on the real-time behavior** !

Many methods of **classical software development** of non-real-time systems can not be used due to the lack of predictability.

For the development of real-time systems, special methods are required.

1.2 Requirements



(a) logical correct, but timing incorrect

(b) logical correct, and timing correct

The validity of an operation of a real-time system depends on

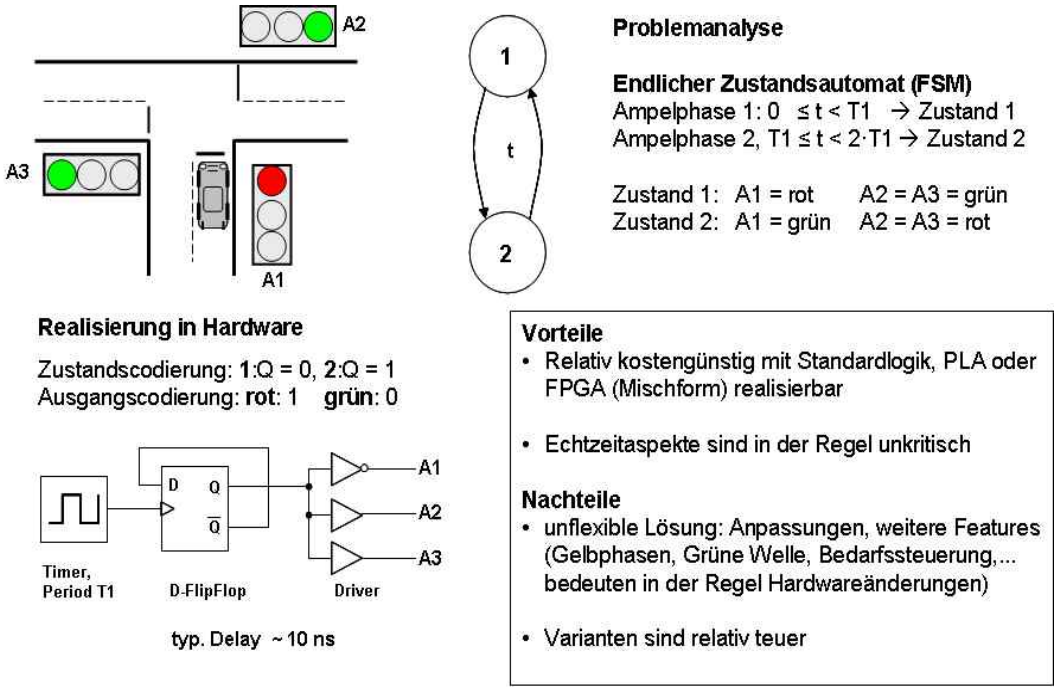
1. its logical result, and
2. the physical time this result is available

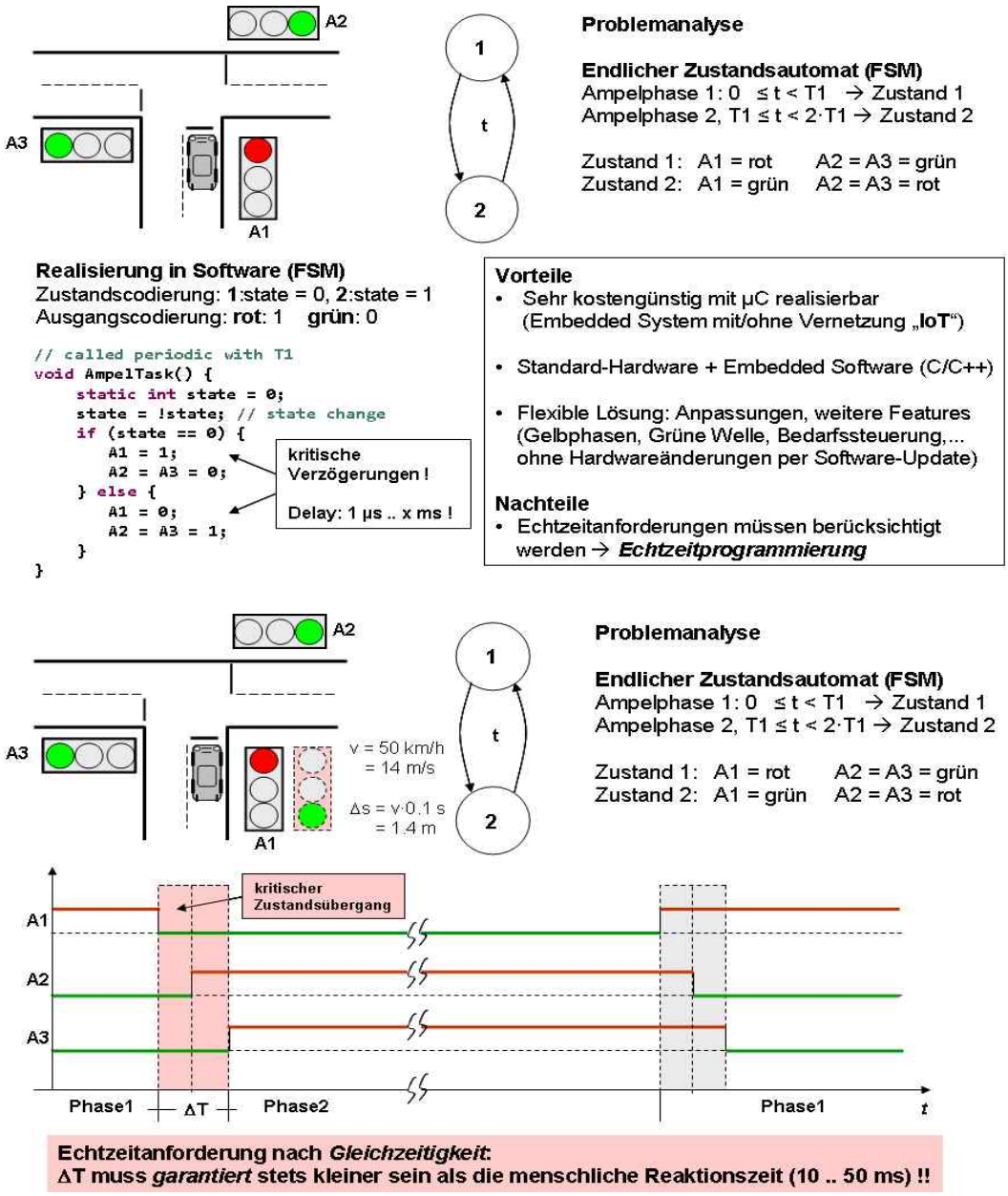
The traffic light control must determine traffic light phases logically and temporally correct.

For real-time systems, the logical correctness **and** the timing correctness is required.

- **Non Real-Time Systems**: logical correctness **OK**
- **Real-Time Systems**: logical correctness + timing correctness **OK**

Example: Real-Time Requirements with a Traffic Light Control





Solution: Critical Section (Mutex) in AmpelTask: see FSM Exercise

```
// called periodic with T1
void AmpelTask() {
static int state = 0;
state = !state; // state change
  begin(mutex);
if (state == 0) {
  A1 = 1;
  A2 = A3 = 0;
} else {
  A1 = 0;
  A2 = A3 = 1;
}
  end(mutex);
```

A Definition of Real-Time Systems is given by DIN 44300 [1985]

Realzeit-Systeme beziehungsweise Echtzeitsysteme sind Computersysteme, dieim Realzeitbetrieb arbeiten.

Realzeitbetrieb wird definiert als der Betrieb eines Rechensystems, beidem Programme zur Verarbeitung anfallender Daten ständig betriebsbereitsind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenenZeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nacheiner zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunktenanfallen.

Wikipedia:

In computer science, **real-time** computing (RTC), or reactive computing, is the study of hardware and software systems that are subject to a ”real-time constraint”, i.e., operational **deadlines** from **event to system response**.

By contrast, a **non-real-time system** is one for which there is **no deadline**, even if fast response or high performance is desired or preferred.

The needs of real-time software are often addressed in the context of real-time operating systems, and synchronous programming languages, which provide frameworks on which to build real-time application software.

A real time system may be one where its application can be considered (within context) to be mission critical.

The anti-lock brakes on a car are a simple example of a real-time computing system - the real-time constraint in this system is the short time in which the brakes must be released to prevent the wheel from locking. **Real-time computations** can be said to have **failed if** they are **not completed before their deadline**, where their deadline is relative to an event.

A real-time deadline must be met, regardless of system load.

Thus, in a real-time system the logical correctness (functional correctness) of the answers with guaranteed response times (temporal correctness) is of essential importance.

The temporal behavior in real-time systems is part of the system specification and thus subject of the product verification.

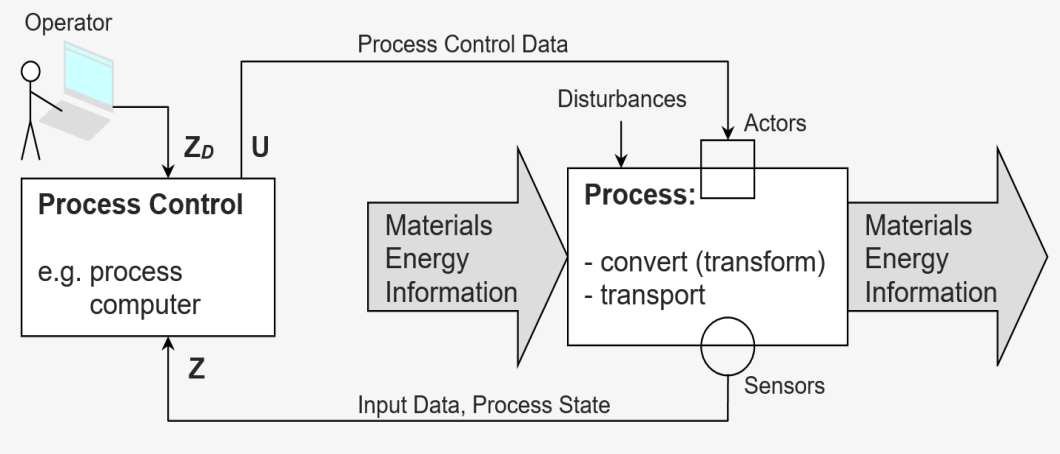
Embedded systems often have to meet **real-time requirements**.

Technical Process and Process Model

A technical process can be defined as follows (DIN 66 201):

In a technical process materials, energy or information (the process media) are converted or transported (such as an end product) into grafted materials, energy or information.

Its input and state variables summarized in the vector **Z** can be measured with sensors and controlled by actuators:



In automation, the process is controlled via control actions (vector **U**) so that after a given time, a desired process target or a desired state $\mathbf{Z}_D$ is reached.

The desired state can refer to the following:

quality, quantity (of the final product), speed, stability, security (during the transforming process), production, yield, raw material consumption, emissions, or even reaching the next process step (e.g. with sequential processes).

The delay time in the loop is of ***crucial importance*** for the ***stability*** of the ***entire system***!
This delay time must therefore be exactly specifiable ***Real-Time Programming*** !

Timeliness The timeliness (=timing correctness) is directly derivable from the above considerations requirement for all real-time systems. For digital process computers timing correctness is, the output data must be calculated in time.
This also requires that the input data must be collected on time. The timing deadlines, and are usually given by a technical process:
Thus the allowed response time for the controller is limited to a known value, thus real-time systems are said to be **predictable** or **deterministic**.
Input data must be read in time for a real-time system, and the output data to be generated must be provided within the given deadline (time condition).
The primary requirement for a real-time control is the **constant ability to input, process and output data on time regardless of the system workload**.
For a real-time system also means that it **monitors the temporal validity** of information and prevents the use of invalid data !
This must apply regardless of whether the data to be processed are **event based** or at **predetermined (e.g. periodic) times**.
However, due to complexity, the timing behavior of real-time systems **can never be predicted with 100 % certainty** System validation by tests !

Time Conditions The **time conditions** to monitor, and control a process can be in several variants

1. Precise Time, Exact Time A precise time t_0 for a controller action is defined. This action must not be executed earlier or later. **Examples**: Sampling Systems (DSP & Digital Control), clock display control.

2. Latest Time Limit, DeadlineA maximum time t_{max} (=deadline) for a controller action is defined. This action must be finished latest at its deadline, but it can be finished earlier. **Examples**: a sampling system calculating an output sample for the next sampling period, Anti-Squeeze Control, maximum response time with switching on/off a machine, UI control reaction time to a button, slider, ... e.g. $t_{max} < 50$ms with human reaction times "*immediately*"

3. Earliest Time Limit A minimum time t_{min} for a controller action is defined. The action must not be executed earlier, but it can be executed later. **Examples**: transitions in state machines, an output may not occur before a state transition has been finished.

4. Time IntervalAn action must be executed within a time interval $[t_{min}, t_{max}]$, thus at any time earlier than t_{max} and later than t_{min}. **Example**: airbag ignition in some crash situations an airbag shall be ignited e.g. 10-30 ms, in other situations 5-8 ms after crash.

Time conditions can be **periodic** or **non-periodic**.

Periodic Time Condition

A periodic time condition is given with analog signals to be sampled and processed by a digital computer. Any analog signal has to be discretized in both value and time by a constant periodic sampling time T_s. This is typical for sampling systems realizing digital filters or digital control algorithms for processing quasi analog signals.

An example is the electric signal of a microphone which has to be sampled **strictly periodic** in order to have a **digital representation** of the **original analog sound signal**. Any deviation from the periodic sampling results in unwanted noise, which cannot be compensated. After time sampling the values of the voltage signal is discretized by an AD-Converter.

Sampling Theorem [Kamm], [Oppen]

Example: CD-Audio: $f_s = 44.1$ kHz, $T_s = 22.7$ µs $B_x = 20$ kHz $< 0.5\, f_s = 22.05$ kHz

Correct sampling: tagesschau_44.1.wav Sampling theorem violation: tagesschau_D10.wav

Digital signal processing of analog signals involves very strict requirements to real-time conditions, due to the periodic and exact sampling conditions.

Action	**Time Condition**
input of a sample of an analog signal	exact periodic sampling at $k \bullet T_s$
computation of a signal sample	time interval $[k \bullet T_s + T_{ADC}, (k+1) \bullet T_s]$
output of the calculated signal sample	exact periodic sampling at $(k+1) \bullet T_s$

Non-Periodic Time Condition

A non-periodic time condition is given frequently with digital signals to be processed at arbitrary, non-predictable times due to events like pushing or releasing a switch or a revolution sensor, producing a slope at discrete angles of a motor shaft.

Absolute Time Condition With an absolute time condition defined, an action must be executed at a certain global time, i.e. an action must be performed at 12:00:00 MEZ.

Relative Time Condition With a relative time condition defined, an action must be executed relative to some previous event. Example: a control signal must be updated after 0.5 s after a push button was pressed.

All time conditions can arise in combinations.

For meeting the time conditions a real-time system must have

1. **sufficient processing speed** to process the input and output data at the required speed (sampling frequency), which the technical process requires

2. **deterministic time behavior**

Hard and Soft Real-Time Systems In the previous section, we saw that computation must complete before reaching a given deadline. In other words, real-time systems have timing constraints and are deadline-driven. Real-time systems can be classified, therefore, as either hard real-time systems or soft real-time systems.
What differentiates hard real-time systems and soft real-time systems are the degree of tolerance of missed deadlines, usefulness of computed results after missed deadlines, and severity of the penalty incurred for failing to meet deadlines.
For **hard real-time systems**, the level of tolerance for a missed deadline is extremely small or zero tolerance. The computed **results after** the **missed deadline are** likely **useless** for many of these systems. The penalty incurred for a missed deadline is **catastrophic** (e.g. an unstable control loop).
For **soft real-time systems**, however, the level of tolerance is non-zero. The computed **results after** the **missed deadline** have a **rate of depreciation**. The **usefulness** of the results does **not reach zero immediately** passing the deadline, as in the case of many hard real-time systems. The physical impact of a missed deadline is **non-catastrophic**.

1. **Hard Real-Time Conditions** the time conditions must be met, or catastrophes occur ! Deviations from the time-conditions are not allowed and represent a serious error. In safety-related systems a violation of the hard real-time time conditions can endanger human life (e.g. if sampling controls become unstable). Systems, meeting the hard real-time conditions are called ***Hard Real-Time Systems***. **Examples**: Sampling Controls (closed-loop), aerospace flight controls, airbag ignition, automotive X-by-wire, pacemakers, ...

2. **Soft Real-Time Conditions** the deadlines must be met but with a degree of flexibility. The deadlines can contain varying levels of tolerance, average timing deadlines, and even statistical distribution of response times with different degrees of acceptability. A missed deadline does not result in system failure, but costs can rise in proportion to the delay, depending on the application. Systems meeting soft real-time conditions are called ***Soft Real-Time Systems***.**Examples**: multimedia-streams (speech, video, music, low level audio), toasters, refrigerators, ...

Concurrency Real-time systems must generally treat multiple inputs and outputs simultaneously. Thus, the simultaneous addition to the timeliness of the second general requirement for real-time systems.
Example is about a CNC machine tool, where the x-y axes must be controlled simultaneously (synchronously, concurrently) to move the tool along a predetermined path $B(x, y) = B(x(t), y(t))$.
There is an relative time condition for both processes acting concurrently on the two drive axes x and y:
If the time axis of x and y are shifted by some amount of time ΔT (e.g. due to different clocks for x- and y-drive control), the resulting path $B(x, y) = B(x(t), y(t\text{-}\Delta T))$ will be wrong.
Example: Tolerances $|x|, |y| < 0.05$ mm, Requirement $v_x = v_y = x' = y' = \pm 10$ cm/s Requirement $|t| < |x| / |v_x| = 0.5$ ms Synchronization, Real-Time Condition
Realizations of Real-Time Systems with Concurrency Requirement
To meet the requirement of concurrency, there are several possibilities:

1. Full parallel processing in a multiprocessor system

2. Quasi-parallel processing in a multiprocessor system

3. Quasi-parallel processing in a uniprocessor system (**Multitasking**)

1. Each task is processed on a separate processor (e.g. microcontroller). There is a real parallel processing, each task has the full power of its own processor. This allows the independent consideration of each task, however, the tasks must be synchronized in most cases (e.g. x-y axis control of the CNC machine).

2. Each task executed on a processor that is allocated by a real-time scheduler(there are usually fewer processors available than tasks). The real-time scheduler is a hardware or software component, which assigns the tasks to processors such that all tasks can meet their timing constraints.The individual tasks can be assigned to single processors.

3. From a hardware perspective, the simplest and most cost effective and realization: All tasks share a single processor Multitasking. A Real Time Scheduler controls the allocation of the processor to the task. This form of real-time scheduling is the easiest to analyze and control preferred realization of most embedded systems.

Availability Real-time systems must be available without interruption, otherwise time conditions may be violated. This leads to the third basic requirement of real-time systems, the **availability**.
Real-time systems must be available over a longer period of time, maybe around the clock without any breaks, as with

1. Power Plant Controls

2. Production Facilities

3. Heating and Air Conditioning Systems

4. Communication

5. Medical Applications

6. ...

This requires that there is no disruption of operations for phases of system reorganization. A typical example here is the garbage collection of some programming languages such as Java or C # or their runtime environments. For example, the default Java implementation is not suitable for use in real-time systems. Remedy is the use of algorithms free of need for reorganization (such as for dynamic memory management).
Example: Algorithm for Calculation of a Factorial in C OK with Real-Time programming Not OK with Real-Time programming ! (Exec.-Time of stack operations not deterministic due to recursion).
Another possibility is to divide the reorganization into small steps under control of the real-time scheduler to ensure that no time constraint is violated: This method is been used by Real-Time Java (Real-Time Specification for Java (RTSJ)).Also, more traditional memory management strategies involve not predictable time for execution (like **malloc/free** of the C standard library, **new/delete** with C++) dynamic memory management is typically avoided with hard real-time systems !
Example:
Not OK with Real-Time programming OK with Real-Time programming !(Exec.-Time of malloc() non deterministic)
- but: malloc() in initialization possible

1.2.1 Timing Definitions

Process Time and Processing Time
The time interval between two requests of the same type is called the **process time** T_P. If the interval between two events is constant, it is called a periodic event.
In many cases, however, the time interval varies, so that there is a time interval with a minimum T_{Pmin} and a maximum T_{Pmax}.
For the actual processing of the event computer instructions (operations) have to be executed. The event belongs to the sequence of instructions is referred to as the code sequence or a job. Jobs are implemented within the computer as a computational process, which can be distinguished as **tasks** and **threads** (see section **Process Management**).
For the execution of a job i the computer needs **processing time** T_{Vi} (from de: *Verarbeitungszeit*) which is also called **execution time**.
The **processing/execution time** needed for a certain number of operations N is inverse to the **computing power** P (in ops / sec) a processor core (CPU) offers:

	$T_V = N / P\ [T_V] = \text{ops} / (\text{ops} / \text{sec}) = \text{sec}$	(1.1)

The CPU power is approximately proportional to the CPU clock frequency.
In practice it is not easy to specify the processing/execution time for a certain task, since it often depends on the input values (conditional branches). Furthermore, the performance of the processor varies (due to caches, pipelines and DMA). To calculate T_{Vmax} in real-time systems, N_{max} and P_{min} shall be used in (**??**).
Timeliness, Response time and Maximum Response Time
Timeliness means, that a task required at the time t_0

1. is done **not before** a specified time $t_0 + T_{Zmin}$ and
2. is done **no later** than a specified time $t_0 + T_{Zmax}$ (timeliness).

The requirement for a minimum latency, i.e, that a task is not been done before a certain time, is often missing ($T_{Zmin} = 0$) or can be realized easily. On the other hand, the requirement for **timeliness** ($T_R < T_{Zmax}$) is hard to guarantee. The **maximum reaction time** T_{Rmax} is the worst-case time when task execution ends and the computer can cause a reaction. The **maximum reaction time** T_{Rmax} always needs to be **smaller** than the **maximum response time** T_{Zmax}.
Utilization, Load
The computer core (CPU, processor) is loaded (utilized) depending on the frequency of occurrence of an event. The utilization gives a relative measure how much the CPU is loaded by a task is the quotient of the necessary processing time T_V and process time T_P

	$= T_V / T_P$	(1.2)

The total utilization $_{ges}$ is the sum of the utilization of all individual jobs (**??**). A real-time system must be able to process all tasks with fulfilling their requirements for timeliness – this is sometimes referred to as the requirement for concurrency (in a wider sense). Mathematically, this means that the total utilization $_{ges}$ is less than 100%:

	$\rho_{ges} = \sum_{i=1}^{n} \rho_i = \sum_{i=1}^{n} \frac{T_{Vi}}{T_{Pi}}$ $\rho_{ges} < 1$	(1.3)

1.2.2 Real-Time Evidence

Evidence that all time requirements of all requests are met under all conditions, is very difficult (practically impossible). In principle, the real-time evidence is based on

1. evaluate the relevant characteristics of the technical process,

2. number of different requirements
3. Minimum processing time for each request
4. Minimum allowable response time for each request T_{Zmin}
5. Maximum allowable response time for each request T_{Zmax}
6. dependencies between events
7. identify the maximum processing time T_{VMax} (WCET = Worst Case Execution Time) for each request
8. check the **load condition** (**??**)
9. verify the **timeliness condition** (determination of T_{Rmin} and T_{Rmax}).

With verifying the **timeliness condition** in particular, it is not trivial to determine the maximum response time T_{Rmax}.

Estimate for the Worst Case Execution Time (WCET)

The maximum processing time T_{VMax} – often called Worst Case Execution Time (WCET) – of a request is very difficult to determine. The WCET depends in particular on the algorithm, the implementation of the algorithm, the hardware used and the other activities of the system (other computing processes). Therefore, the WCET at best be estimated. In principle, two methods for determining the WCET can be distinguished:

1. measure the WCET
2. static code analysis with WCET determination (count operations,cycle).

WCET Measure

The WCET is determined by execution of the code sequence that will be processed, normally on the target platform. The time between the onset of the event and output at the end of the code sequence is determined. Condition: Modification of the real-time software

Method for Measuring WCET

To measure this time can be divided into two basic methods:

1. 1st Measurement by the measure out piece of code itself (see above).The piece of code will be amended such that every time an event arrives and when the reaction took place, a time stamp t_1 (free running timer) is stored. At the end of the code, a second time stamp t_2 is stored. the difference $T_{Vi} = t_2 - t_1$ is the actual execution timeThe first value T_{V1} is stored as T_{Vmax}. With subsequent iterations, a newly calculated time T_{Vi} is compared with the stored value. If the newly calculated time is greater than the previously measured WCET, the new value overwrites the existing WCET$T_{Vmax} = \max(T_{Vi})$
2. External measurement of event and response (output), by means of an oscilloscope using port-toggle.

For a WCET measure, appropriate input values and events must are generated. In addition, the entire system must be put under load (e.g. by an automatic tester).

Advantages of the method:

1. Independent of a programming language

2. Relatively easy to implement

3. Can run as autonomous self-diagnosis

Disadvantages of the method:

1. The WCET of a code sequence can not be guaranteed, after all, it depends on many factors (background, loop iterations, caches, branches, etc ..).

2. The measurement is WCET is time consuming and ultimately expensive. Measure out the piece of code needs to be theoretically charged with all input data (in any combination).

3. This method can be carried out with the running code on the target platform only. Thus, an assessment at an early stage of development can be difficult.

4. A test environment is required (which provides the input data to create).

5. The code sequence to be measured out must be modified (for storing time stamps) timers.

6. It is sometimes difficult to put the system under the required load.

Static Code Analysis
Here, the code itself is analyzed. Therefore most often an analysis tool program is used, which is fed with the object (machine) code, rather than with the C-source code. In addition, a hardware description is necessary (e.g. clock frequency of the microcontroller).
The tool allows the duration of each code sequences to be estimated. After that, the number of loop iterations are considered, a bound of the number of cycles and the estimated execution time T_{SA} is printed.
Analysis can get rather complex, if cache hits, cache misses, processor pipelining are conssidered.
If there are safety requirements, the code measured is not to changed after a static code analysis.
As a safety margin factors of $c = 2, 3$, sometimes are used to state an upper limit
$t_{Vmax} \leq T_{SA} \cdot c$
A manual determination of the number of cycles can be done in simple cases by inspection of the C-compiler generated assembler code:
Example: timer-interrupt routine for AVR microcontroller (e.g. ATmega8_sum.pdf). The C-compiler generated assembly-/machine-code, ultimately determines the number of cycles to complete a section of code for a particular microcontroller (AVR here). The number of cycles multiplied by the instruction cycle time is the execution time.
Estimation of the Best Case Execution Time (BCET)
The determination Best Case Execution Time of a job can be done as the WCET, with the least possible load on the system, which can be easily implemented in general-at least in a test environment.
$T_{Vmin} = \min(T_{Vi})$

1.3 Real-Time Operating Systems

An **RTOS** (Real-time Operating System) must meet the same requirements as a standard general purpose operating system (GPOS) and offers services for

1. **Task Management**This is the management and organization of the implemented programs to be processed, also called tasks. The mission of the task management is thus essentially in the allocation of the processor (or the processors in a multiprocessor system) to the task.

2. **Resource Management**Tasks need resources for their execution, their allocation is the task of resource management. This mainly includes:- Memory management, responsible for allocating memory- Input/Output (I/O) management responsible for the allocation of I/O devices to the tasks.

3. **Communication**The communication between tasks, called inter-process communication.

4. **Synchronization**A special form of communication is the synchronization, which refers to the timing of the tasks.

5. **Protection**The protection of resources against unauthorized access by tasks.

These traditional requirements are exactly the same RTOSes as with GPOSes. Depending on the application, some of the services might be implemented only rudimentary or completely absent. This is in particular with **embedded systems**, where the **RTOS** is kept as **lean as possible**, due to the limited resources.
In addition to the traditional OS requirements, real-time operating systems are required to

1. Respect for timeliness and concurrency,

1. Respect for availability.

These requirements of RTOS dominate the other requirements, even if compromises are necessary.

1.3.1 Structure of an RTOS

A real-time operating system (RTOS) is a program that schedules execution in a timely manner, manages system resources, and provides a consistent foundation for developing application code. Early operating systems had a monolithic structure, i.e. all functionality was implemented in a uniform, not further subdivided software block. This led to a number of disadvantages such as poor maintainability, poor adaptability and high error rate. Today's operating systems therefore follow a hierarchical layers model.

1. **Device Driver**: This layer abstracts from the hardware, and- realizes the **hardware-dependent control** of each device - realizes the **hardware-independent** interface for the above layer. Ideally, the device drivers are the only hardware dependent layer in the OS. When adapting to other devices, only the device driver layer must be changed.

2. **I/O (Input-/Output) Control**, realizes the **logical**, **hardware-independent** device control.

3. **Resource Management**: responsible for (allocation) and de-allocation (release) of memory and I/O resources.

4. **Task Management**: responsible for the allocation of the processor to each task.

5. **API (Application Program Interface)** realizes the interface to the application.

The OS kernel is critical for the stability and security, it is executed in the so-called ***kernel mode*** of the processor, which allows full access to all resources.
The ***kernel mode*** is a special operating mode of (advanced) microprocessors, enabling privileged instructions, direct access to memory and I/O, change configuration registers, etc.
In normal ***user mode*** this privileged commands are blocked so that an application can not interfere with important operating system parts. This is one of the protection services of the operating system.

In the previous layer model, the core operating system extends over the layers 1 – 4. Since the core contains many layers, it is called a **macro core operating system**.
Today's RTOSes must be highly configurable, especially in the field of **embedded systems** where scarce resources are typical FreeRTOS.
It is therefore desirable to remove unwanted parts from the operating system, e.g. not needed scheduling or protection methods. This leads to the concept of the **micro-kernel operating system**.

1.3.2 Task Management

The task management is a core task of operating systems. The most significant differences between standard and real-time operating systems are found here.
The tasks in a real-time application must meet the requirements for timeliness and concurrency. This requires scheduling strategies for RTOSes different from those found in standard operating systems.

Task Model A **computational process** or **process** (also called **task**) is running as a computer program together with all the variables (including register states) and resources each process has a **main()**
A task is a process controlled by the RTOS for execution of a sequential program. Several tasks are being processed by the quasi-parallel processor, necessary changes between tasks are made by the RTOS' task scheduler.
Changes between tasks are needed to meet all tasks requirements for timeliness.
Within a ***process***, there can be parallel ***threads*** (running simultaneously).
A ***thread*** must share its resources with other ***threads*** of the same ***process*** just 1 **main()**

1. A **task** is called a **heavyweight process,** if it contains its own variables and resources separated from other tasks by the OS. It has its own address space and can communicate with other tasks via interprocess communication. A task realized as a process provides maximum protection, possible interference by other tasks is limited to predefined channels. A change of the processor to another task (**context switch**) is due to the separate resources, which is a time-consuming task.

2. A **task** realized as a **thread** is called a **lightweight process** that exists within a single process. It uses the variables and resources of the process. All threads within a process **share the same address space**. Communication can take place over any global variable within the process. Threads can interfere with any other thread within a task. Shared memory allows great efficiency. Communication between threads is more direct and faster Context switch can take place very quickly, e.g.: FreeRTOS, VxWorks Low data protection between data of individual threads

To fulfill the **real-time requirements**, efficiency is usually more important than protection.
Many real-time applications use **threads within a single task**.
Often, a real-time application is realized by a ***single process***, which contains threads to realize numerous tasks. **Embedded systems** are often based on the **thread concept**, due to scarce resources.
From the perspective of the real-time conditions (timeliness, concurrency, availability), **threads** aren't different from **processes**, both are usually identified by the term **task**.

Multitasking, Context Switch Multitasking is the ability of the OS to handle multiple tasks within set deadlines. An RTOS kernel might have 2 tasks that it has to schedule to run. At certain times, execution of Task 1 has to switch to Task 2

Task States For each task (or threads) one can define 6 different states:

1. **existent**
2. **ready (**waiting)The task is ready, all conditions are fulfilled, resources are allocated, the task is waiting for the allocation of the processor.
3. **running** (executing)The task is run on the processor. In a uniprocessor only one task can be in this condition, in a multi-processor system, several tasks can be in this state simultaneously.
4. **blocked** (blocked)The task is waiting for an event (e.g. an input value, an inter-process communication object) or the release of a resource (task synchronization).
5. **suspended** (finished)A task is suspended from its normal operations by another task. It can be resumed later
6. **not existent**

1.3.3 Real-Time Scheduling

The main job of an RTOS task management is the allocation of the processor to the ready tasks. There are different strategies, so-called scheduling strategies.
A **real-time scheduler** must divide all ready (runnable) tasks to the processor, so that all all time conditions are met. The set of tasks managed by the real-time scheduler is called **taskset**.
For the evaluation of various scheduling strategies, the processor demand H (=**utilization**) is an important quantity for the load of the processor, it is defined as

	$H =$	(2.1)

Each CPU can be utilized up to 100 % at maximum. **Example**: A periodic task 1 with a period of $T_{P1} = 200$ ms and an execution time of $T_{e1} = 100$ ms causes a processor demand of $H_1 = 50$ %
A second periodic task with a period of $T_{P2} = 100$ ms and an execution time of $T_{e2} = 50$ ms also causes a processor demand of $H_2 = 50$ %.
An implicit timing condition for each periodic task is: task execution must be finished before the next period starts (deadline) !
Both tasks cause a total processor demand $H = H_1 + H_2 = 100$ %.
One possible schedule for executing both tasks is to displace task 1 after exactly half of its execution time by task 2, which is never displaced. Thus, task 1 must be displaceable:
another possible schedule:
In general, for a tasklet of n periodic tasks, the total processor demand

	$H = \sum_{i=1}^{n} \frac{T_{ei}}{T_{pi}}$	(2.2)

Classification of Scheduling Algorithms Static Scheduling
Prior to the execution of a taskset an allocation table (dispatching table) with the start times exists, regarding time constraints and dependencies of each task. The dispatcher as part of the RTOS kernel assigns the individual tasks due to this table
principle of synchronous programming.
Advantage minimal overhead, as no decisions are needed at runtime.
Disadvantage restriction to periodic events.
Dynamic Scheduling

Here for the execution of tasks different criteria are used by the dispatcher **at runtime**, taking into account start times, time conditions (deadlines), and dependencies of each task. Asynchronous programming. Advantage increased flexibility and the opportunity to respond to aperiodic events. Disadvantage increased overhead, lower predictability

Static and Dynamic Priorities

(Not to be confused with static/dynamic scheduling). The scheduler can use priorities for the allocation of resources (processor, memory, I/O,...) to a task. **Static priorities** are set before running the application and are never changed during runtime. **Dynamic priorities** can be adjusted for each task **at runtime**. Furthermore, there are algorithms, using **No Priorities** at all.

Preemption (= "Prioritization") is an important feature for a scheduler, which describes the capability for displacing a running task for later execution in favor of another task.

Preemptive scheduling means that a less important task can be displaced by a more important task. The most important task with ***ready***-state will be executed immediately. The less important task will be continued again until no other more important task waits with ***ready***-state.

(a) Preemptive Scheduling (b) Non-Preemptive Scheduling

Non-preemptive scheduling (= **cooperative scheduling**) means, that there is no displacement of a running task. Only after the current task is finished or blocked, the next task with ***ready*** state is executed.

Time-Slice Scheduling (Round-Robin Scheduling) Time-Slice Scheduling (time slicing) assigns each task a fixed time slice (Time Slice). The order of task execution corresponds to the sequence of ***ready***-tasks entering the task-list of the OS scheduler (FIFO principle). The duration of the time slice for a task can be set individually.

This type of time-slice scheduling is a dynamic preemptive scheduling.

TSS does not use priorities

time slot duration can be chosen individually for each task

With time slices chosen fine-grained enough TSS is approaching optimality.

Principle of Time-Slice (Round-Robin) Scheduling:

1. All ***ready***-tasks are queued in a FIFO
2. Each task *i* gets assigned a time slice (time slice, quantum) T_{TSi}
3. If a task after the expiry of its quantum is still ***running***,
   (a) the task is preempted, i.e. displaced to the ***ready*** state
   (b) the task is put at the end of the FIFO queue;
   (c) The first task in the FIFO will be executed.
4. If a task changes state from ***blocked*** to ***ready***, it will be put at the end ofthe FIFO queue

Typical Applications: **Kernel-mode programs Example 1:** 3 Tasks with $H_{max} < 0.778$, from [WörnB] , same as in section 2.2.7: Task T1: period T_{p1} = 10 ms, execution time T_{e1} = 1 ms H_1 = 0.1Task T2: period T_{p2} = 10 ms, execution time T_{e2} = 5 ms H_2 = 0.5Task T3: non-periodic, period T_{p3} = deadline T_{d3} = 15.4 ms, T_{e3} = 2.62 ms H_3 = 0.17

By (**??**) the total CPU utilization H = 0.1 + 0.5 + 0.17 = 0.77

One chooses a basic time-slice T_{TS} = 1 ms and assigns individual time slices as multiples of T_{TS} with ΣT_{TS} = 9 ms:

Task T1: T_{TS1} = 1 .T_{TS} = 1 ms $H_1 = T_{TS1}/\Sigma T_{TS}$ = 1 ms / 9 ms = 11 %

Task T2: $T_{TS2} = 5\ .T_{TS} = 5$ ms $H_2 = T_{TS2}/\Sigma T_{TS} = 5$ ms / 9 ms = 55 %
Task T3: $T_{TS3} = 3\ .T_{TS} = 3$ ms $H_3 = T_{TS3}/\Sigma T_{TS} = 3$ ms / 9 ms = 33 %
(a)
In this example the assigned time slices were chosen to be larger than the execution times of each task, in order to avoid preemption.
With TSS a context-switch can occur at the end of a time-slice only. which can cause some *jitter* (as with task T1 above).

Fixed Time-Slice-Scheduling FT scheduling is based on a TDMA approach, requires no priorities, and is therefore frequently used in embedded microcontroller applications without an operating system.
A strictly periodic schedule is created with a periodic time $N{\cdot}T_{TS}$, i.e. with N integer multiples of the basic T_{TS} time slice. Each task gets assigned one (ore more) basic time slots, large enough to hold the WCET of each:
Conditions for Events are polled within each task, if a task is not finished at the end of its timeslice, it can be preempted (***preemptive FTS***)or not (***cooperative FTS***).
The previous example with three tasks, and an idle task T4 at a period $N{\bullet}T_{TS} = \Sigma T_{TS} = 10$ ms is realized, the time slices are chosen as multiples of $T_{TS} = 1$ ms as follows
Task T1: $T_{TS1} = 1\ .T_{TS} = 1$ ms $H_1 = T_{TS1}/\Sigma T_{TS} = 1$ ms / 10 ms = 10 %
Task T2: $T_{TS2} = 5\ .T_{TS} = 5$ ms $H_2 = T_{TS2}/\Sigma T_{TS} = 5$ ms / 10 ms = 50 %
Task T3: $T_{TS3} = 3\ .T_{TS} = 3$ ms $H_3 = T_{TS3}/2\Sigma T_{TS} = 3$ ms / 10 ms = 30 %
Task T4 (idle) $T_{TS4} = 1\ .T_{TS} = 1$ ms $H_4 = 10$ %
T3 runs for the second time at $t = 26$ ms the delay of 10 ms does not result in a T3-deadline violation.
The idle task T4 provides a placeholder for unused CPU load, which can be used in modern CPUs for power saving features.
Advantage of fixed TSS: the execution times can be defined exactly!

Binary Fixed Time-Slice-Scheduling If task periodic times T_{Pi} can be represented as power-of-2 integer multiples of a basic time slice T_{TS}, one can obtain a periodic binary schedule:
If the maximum time of execution is T_{TS} and equal for each task, the utilization for each task is

$$H_i = \frac{T_{ei}}{T_{pi}} = \frac{T_{TS}}{2^i \cdot T_{TS}} = 2^{-i} \sum_{i=1}^{N} H_i \overset{N\to\infty}{\longrightarrow} 1 \qquad (2.3)$$

Thus, the fastest task with periodic time $T_{p1} = 2.T_{TS}$ is assigned 50 % of the complete processing time, task 2 with periodic time $T_{p2} = 4.T_{TS}$ gets assigned 25 %, task 3 gets 12.5 %, ...
From (**??**) it shows, that the total utilization of H aiming with an infinite number of tasks to 100%, thus, there will be no overload situations, if all tasks meet their maximum execution time T_{TS} requirement (easily guaranteed with a preemptive schedule).
$T_{TS} \geq \max(T_{e1}, T_{e2}, T_{e3}, ..)$
With a non-preemptive schedule (cooperative task schedule) maximum execution times T_{ei} have to be determined carefully (WCET determination), such, that each task can complete execution within the basic T_{TS}. If a task gets delayed for some reasons (like high priority interrupts during execution) the complete schedule will get delayed.
Advantages
Binary fixed time-slice scheduling is

1. quite suitable with multirate sampling systems (e.g. oversampling, when integer multiples of a common sampling frequency are needed).
2. a very simple scheduling without priorities,
3. suitable in a non-preemptive realization for even small microcontrollers without OS.

4. Time and duration of execution for each task is guaranteed (apart from interrupts)

Disadvantages

1. Binary time-slice scheduling is not fully flexible with arbitrary periodic times and execution times

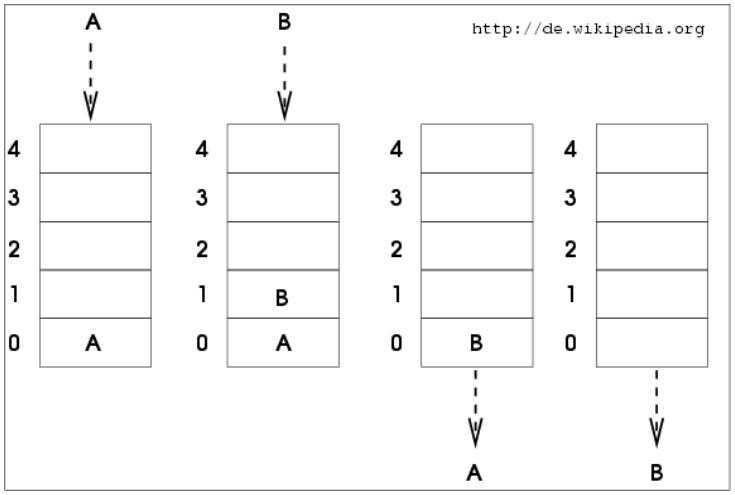
2. can turn into very complex software, especially with integrating tasks with execution time larger than T_{TS}.

Lab-Experiment AVR-Timer-Interrupts-(FTS-Scheduling)
Sketch for Arduino Nano (FixedTimeSlice.ino)
Example: 4 periodic tasks (on an AVR μC, from section Digital PID Control with 4 Tasks)
$T_{p1} = T_{p2} = 1$ ms, $T_{p3} = 16$ ms, $T_{p4} = 8$ ms
$T_{e1} = 50$ μs, $T_{e2} = 0.3$ ms, $T_{p3} = 40$ μs, $T_{p4} = 40$ μs
$H_1 = 5$ %, $H_2 = 30$ %, $H_3 = 0.1/20 = 0.5$ %, $H_4 = 0.05/20 = 0.25$ % $H = 30.75$ %
Solution with Binary time-slice schedule with 3 Tasks, $T_{TS} = 0.5$ ms, $T_{TS} \geq \max(T_{ei}) = \max(0.3, 0.05, 0.005\ 0.0025$ ms) tasks with the same periodic times run in the same time-slice: T1 und T2 executed successively in the 2 Timeslice $2{\cdot}T_{TS} = 1$ ms

t/T_{Ts}	**16**	**8**	**4**	**2**	**1**
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
7	0	0	1	1	1
8	0	1	0	0	0
9	0	1	0	0	1
10	0	1	0	1	0
11	0	1	0	1	1
12	0	1	1	0	0
13	0	1	1	0	1
14	0	1	1	1	0
15	0	1	1	1	1
16	1	0	0	0	0
17			0	0	1
...					

time-slice indexes
Proof for collision-free assignment of time-slice indexes (with $k,l \in \mathbb{Z}$ integers)
for some period N we get time slices at indexes $n_k = k{\bullet}N + \frac{N}{2}$ e.g.: 2, 6, 10, 14, ..
thus, for period $2N$ (next time slice) we get $m_l = l{\bullet}2N + N$ e.g.: 4, 12, 20, 28, ..
for a collision, we set $n_k \stackrel{!}{=} m_l$. Division by N on both sides shows $k + 0.5 \stackrel{!}{=} 2{\bullet}l + 1$, which has no solution with integers k and l, thus the above assignment is without any conflict all time slice indexes (proof by full induction) !

FIFO Scheduling The name derives from the FIFO (first in, first out) principle of a waiting queue:



where elements put into a queue in a certain order of sequence, are taken out in the same order of sequence.With a **FIFO scheduler**, the tasks are processed in the same order in which they have taken the ready state. An running task is not interrupted, it is a **non-preemptive**, **dynamic** scheduler.
Advantage: very simple implementation, sometimes used in universal OS
Disadvantage: bad real-time performance, violations of time conditions even at low processing demands (example)
Example: Two tasks with FIFO scheduling
Task 1: T_{p1} = 150 ms, T_{e1} = 15 ms H_1 = 0.1
Task 2: T_{p2} = 10 ms, T_{e2} = 1 ms H_2 = 0.1
For both periodic tasks there is a time bound (deadline) with the next period. With (**??**) $H = H_1 + H_2$ = 0.1 + 0.1 = 0.2
For 20 % processor demand only, it should be easy to find a schedule for this taskset, but:
This simple example shows that even at a very low processor demand, a FIFO scheduler **cannot guarantee** compliance with the real-time conditions.

Fixed-Priority Scheduling For fixed-priority scheduling, each task is assigned a fixed priority. The ***ready***-task with the highest priority waiting in the scheduler's queue is assigned to the processor. fixed-priority scheduling is **dynamic scheduling with static priorities**. (often used with interrupt processing of microprocessors, preemptive or non-preemptive):

1. **Fixed-Priority-Preemptive Scheduling (FPP)**If a **task with higher priority** than the currently running gets into the ready state, then the **current task is interrupted (displaced, preemted)** and the task with the higher priority is assigned to the processor immediately.

1. **Fixed-Priority-Non-Preemptive Scheduling (FPN)**If a **task with higher priority** than the currently running gets into the ready state, then the **task is assigned** to the processor only **after the current task** is either **completed** or **blocked**.

Advantage: with FPP, the compliance with the real-time conditions can be guaranteed (unlike to FIFO scheduling), if the priorities were assigned appropriate.
The assignment of priorities to tasks is an important step in the development of a real-time application.
Rate-Monotonic-Scheduling (RMS)
For purely periodic applications, there is a rule, the so-called **rate-monotonic scheduling rule**, which states, that the priority of the tasks to be performed is inversely proportional to their periodic time:

	$PR_i \sim \frac{1}{T_{pi}}$	(2.4)

under the (rather idealized) conditions:

1. Preemptive scheduling is used

2. The periodic times T_{Pi} are constant

3. The time limits (deadlines) are equal to the periodic times T_{Pi}

4. The execution times are known and constant T_{ei}

5. The tasks are independent from each other (can not deadlock)

Example: (same as with FIFO scheduling in section 2.2.6). Two tasks with Fixed-Priority-Preemptive Scheduling (FPP):
Task 1: T_{p1} = 150 ms, T_{e1} = 15 ms, lower priority by (**??**)
Task 2: T_{p2} = 10 ms, T_{e2} = 1 ms higher priority by (**??**)
The running task 1 is displaced with an event for task 2, which has higher priority (its state goes from ***blocked ready***), so that the deadline of task 2 is not violated (in contrast to FIFO scheduling).
Obviously, **FPP scheduling** with **RMS** provides much better results than FIFO scheduling.
In contrast, with **FPN**, the deadlines of task 2 would be violated !
In general
This also means that FPP scheduling with RMS always delivers an executable schedule, if one exists.
However, the overall CPU utilization may not exceed H_{max} ([1], by Liu)

	$H_{\max} = n \cdot \left(2^{1/n} - 1\right) = n \cdot \left(\sqrt[n]{2} - 1\right)$	(2.5)

n	H_{max}
1	1.000000
2	0.828427
3	0.779763
4	0.756828
5	0.743492
10	0.717735
20	0.705298
50	0.697974
100	0.695555
1000	0.693387
10000	0.693171

The limit (**??**) can be used to examine the feasibility of a (periodic) taskset and to guarantee the conformance with all time limits.
FPP with RMS is very popular because of its simplicity. However, there are problems at very high utilization (beyond or near H_{max}) and with RMS at the same or nearly the same time periods, delivering same task priorities. Furthermore, there are sometimes difficulties in approaching non-periodic processes by periodic processes with sufficient precision.
Example 1: 3 Tasks with $H_{max} > 0.778$, from [WörnB] : Task T1: period T_{p1} = 10 ms, execution time T_{e1} = 1 ms $H_1 = 0.1$Task T2: period T_{p2} = 10 ms, execution time T_{e2} = 5 ms $H_2 = 0.5$Task T3: non-periodic, deadline = T_{d3} = 15.4 ms, execution time T_{e3} = 5.5 ms
By (**??**) the total CPU utilization $H = 0.1 + 0.5 + 0.357 = 0.957$, amost 100 % !But with (**??**) $H > H_{max} > 0.78$ deadlines will be violated with **FPP/RMS** with 3 tasks.
FPP/RMS **can't deliver an executable schedule**, as this example shows:
By (**??**) one has 2 choices for assigning the priorities due to the equal periods T_{p1} and T_{p2}:
Priority assignment (a) Priority assignment (b)

T1	high priority
T2	medium priority
T3	low priority

T2	high priority
T1	medium priority
T3	low priority

In both cases there is a violation of the T3 deadline, if all tasks get ***ready*** the same time at $t = 0$:
(a)
(b)
Of course, the violation of T3 deadline in the example above occurs due to the deliberate disregard of the load limit H_{max}, by (**??**). If one meets the max. load requirement $H < 78\%$ (n = 3), there is violation of the T3 deadline.
However, with growing number of tasks the maximum load goes down as low as 70% (from n > 10), which is surprisingly low.
Ideas to achieve higher processor loads lead to dynamic assignment of priorities.
Example 2: 3 Tasks with $H_{max} < 0.778$, from [WörnB] : Task T1: period T_{p1} = 10 ms, execution time T_{e1} = 1 ms $H_1 = 0.1$Task T2: period T_{p2} = 10 ms, execution time T_{e2} = 5 ms $H_2 = 0.5$Task T3: non-periodic, period T_{p3} = deadline T_{d3} = 15.4 ms, T_{e3} = 2.62 ms $H_3 = 0.17$
By (**??**) the total CPU utilization $H = 0.1 + 0.5 + 0.17 = 0.77$ With (**??**) $H < H_{max} = 0.78$ deadlines will be not violated with FPP/RMS with 3 tasks:
By (**??**) one has 2 choices for assigning the priorities due to the equal periods T_{p1} and T_{p2}:
Priority assignment (a) Priority assignment (b)

T1	high priority
T2	medium priority
T3	low priority

T2	high priority
T1	medium priority
T3	low priority

In both cases there is no violation of the T3 deadline, even if all tasks get ***ready*** the same time at $t = 0$:
(a)
(b)
If one meets the max. load requirement $H < 78\%$ (n = 3), there is violation of the T3 deadline.
However, with 3 tasks the maximum load allowed is as low as 77.8% (for $n = 3$), which is surprisingly low.
Ideas to achieve higher processor loads lead to dynamic assignment of priorities next sections.

Earliest-Deadline-First Scheduling (EDF) In Earliest-Deadline-First Scheduling (EDF) the Task Processor is granted to that ***ready***-task, which is closest to its time limit (deadline). This is achieved by assigning the task priorities according to the vicinity to the individual deadlines.
EDF is dynamic scheduling with dynamic priorities, either preemptive or non-preemptive
For **preemptive EDF scheduling** a context-switch is made, when a task with earlier time bound gets ***ready***. In **non-preemptive EDF scheduling** is the task with the shortest time interval ist executed only, if the currently running task is finished or blocked.
Preemptive EDF scheduling, is used more frequently than non-preemptive EDF scheduling, which is essentially only used when non-interruptible processes are to be managed, such as disk access.
Advantage: With **preemptive EDF scheduling** 100 % CPU utilization can be achieved.
Example with 3 Tasks (same as in section 2.2.7): Task T1: period T_{p1} = 10 ms, execution time T_{e1} = 1 ms $H_1 = 0.1$Task T2: period T_{p2} = 10 ms, execution time T_{e2} = 5 ms $H_2 = 0.5$Task T3: non-periodic,

deadline = T_{d3} = 15.4 ms, execution time T_{e3} = 5.5 ms(each periodic time is assumed to be deadline).
Again, by (**??**) the total CPU utilization $H = 0.1 + 0.5 + 0.357 = 0.957$, almost 100 % !
(at $t = 0$, T1 and T2 have equal priorities, due to equal deadlines, in such a case, the scheduler decides randomly, e.g. for the task with the lower id)
It can be seen, that with EDF scheduling an executable schedule is found.
Liu [WörnB] shows, that EDF scheduling is an optimal scheduling with

	$H < 100$ % for preemptive EDF scheduling	(2.6)

Advantage:

1. as long as the processor utilization is less than 100 % with a uniprocessor system, EDF scheduling delivers an executable schedule and compliance with all time conditions is guaranteed.

Disadvantages:

1. increased computational complexity for the dynamic priorities needed at run time.
2. the sequence of task assignment is difficult to control.
3. the time of execution is difficult to control with fixed time requirements.

1.3.4 Task-Synchronization and Communication

As most of the resources may only be used exclusively, tasks that request these resources must be synchronized for exclusive use, sometimes with regard to some sequence of access.
Thus, the OS provides **means for synchronization** either for

1. mutual exclusion, and
2. cooperation (sequence of access).

Synchronization The problem of synchronization of tasks arises when these tasks are not independent
from each other. Dependencies arise, when common resources are be used, then the access needs to be coordinated. Common resources may be

1. **Data**Multiple tasks read and write access to shared variable, like tables. (without synchronization, uncoordinated access could result in inconsistent data, e.g. when task 1 is reading values of a table, while another task 2 is updating that table partly.

1. **Devices** Multiple tasks using common devices such as sensors or actuators. (again, coordination is necessary to not to send e.g. contradictional commands of two tasks to a stepper motor drive).

1. **Programs** Several tasks share common programs, such as device drivers. Competing calls to a device driver must be assured to leave consistent application states.

Example: Two tasks compete for access to a data table:
task 1 is reading several temperature sensors, and stores these values in a table
task 2 is reading this table, and prints out the temperature distribution.
Without synchronization, this can lead to erroneous results if, for example task 1, the common table has not yet been fully updated, while task 2 accesses. Then, task 2 gets mixed new and old temperature values, which can lead to undefined states !
There are two basic types of synchronization:

1. The **Mutual Exclusion** or shortly called ***Mutex***, ensures that access to a common resource is permitted only to one task at a time.

Example: of temperature measurement using the **mutex** synchronization.
For the protection of the common temperature table, a **mutex** is defined, toexclude the possibility that both tasks **access** this resource simultaneously.Before entering the critical section a task tries to acquire the mutex from the OS. If the mutex is already occupied by the other task, the newly accessing task is blocked until the other task **releases** the **mutex** again. The task will **un-block,** acquisition of the **mutex** succeeds, allowing the task to enter the critical section.The sequence, which task 1 or 2 acquires the **mutex** does not matter.

1. Synchronization, where the order of access to common resources matters is called **cooperation** –unlike as with **mutex synchronization**, which doesn't regard the order of task requests.

Both, mutex synchronization and cooperation objects can be realized with semaphores.

Semaphores A semaphore (from the Greek σ μ α = "sign" and φ ε ρ ε ι ν ="carry") is historically a signal mast or flag signal, in Information Technology it is an object for synchronization of processes. A **semaphore** is basically a counter variable and two non-interruptible **operations**:

1. **Acquire** (also called "**take**", "**signal**", "**lock**", or "**Passieren**", (**P**))
2. **Release** (also called "**give**", "**wait**", "**unlock**" or "**Verlassen**", (**V**))

If a task tries to ***acquire*** a semaphore, an associated counter variable is decreased. As long as the counter variable has a value less than 0, then the acquiring task is blocked.
Thus, during initialization the positive value of the semaphore states the number of tasks -waiting in a FIFO queue, or in a list using priorities- that may pass the semaphore, and thus enter the critical region protected by the semaphore.
If a task ***releases*** a semaphore the associated counter variable is increased again. If the value of the counter variable is smaller than 1, tasks trying to acquire the semaphore are blocked, or, if the counter is greater or equal than 1, a waiting task is released from its blocking state the task gets ready.Thus, a negative semaphore counter value indicates the number of tasks that have been prohibited to pass a semaphore. **Binary semaphore** only use **0** and **1** as counter values.
It is of crucial importance that the operations "Passieren" and "Verlassen" (as originally introduced by Dijkstra) are realized atomically, i.e. they cannot be interrupted by any other operation. Only then, a consistent handling of the counter variable is ensured. Semaphore objects are managed by the RT-OS.
Other names for the **semaphore operations**:
P(sem) = Passieren(sem) = sem.P() = sem.decrement() = sem.wait() = take(sem)$_{VXWorks}$
V(sem) = Verlassen(sem) = sem.V() = sem.increment() = sem.signal() = give(sem)$_{VXWorks}$

1. For **mutex-synchronization** a semaphore is used, with counter initialized with 1. Therefore, only one task can pass, gaining access to the protected resource:

Example: Semaphore as Mutex in FreeRTOS

1. A **cooperation synchronization** can be realized by means of two semaphores. Example: sequential order of a task cooperation T2T1T2 with 2 semaphores:

Example: Cooperation in FreeRTOS