Slides to

Real-Time Programming

Prof. Dr. Markus Pfeil

May 1, 2023

HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Contents

1 Real-Time Systems

1.1 Introduction

Real-Time systems are used in the wide area of automation have 2 basic requirements.

- The **logical correctness** of the systems output in response to its inputs.
- 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 with 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 micro-controller applications.

Figure 1.1.1: Automotive Door Contol Unit with Anti-Squeeze

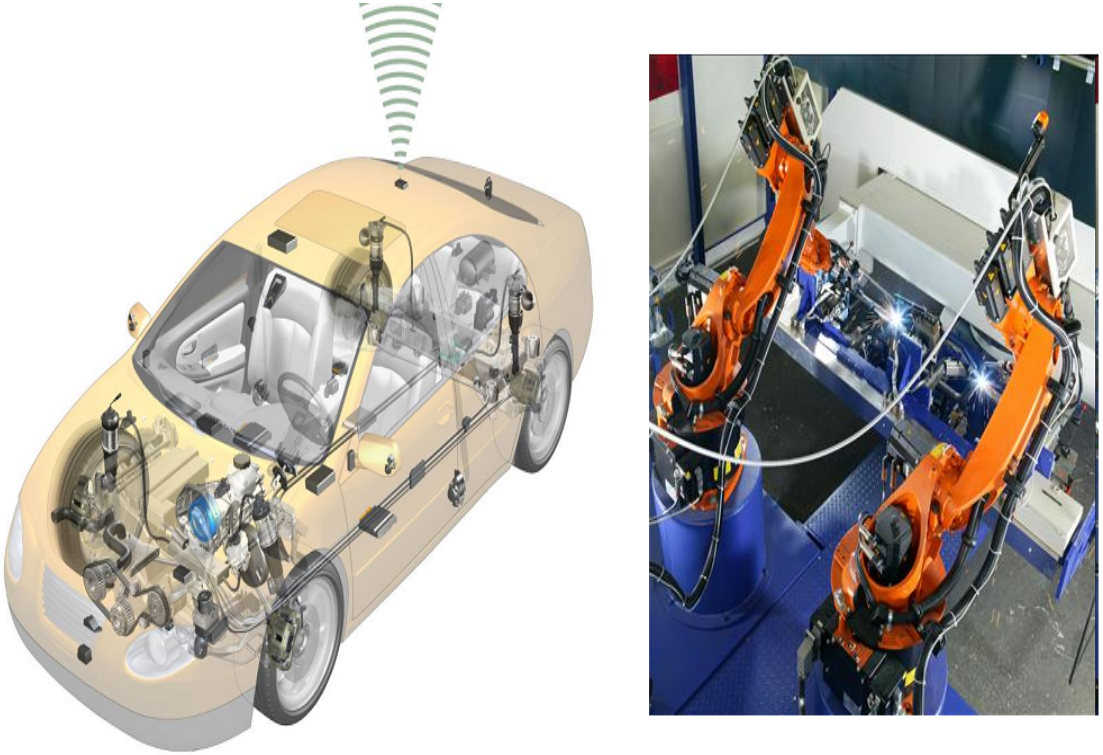


Figure 1.1.2: Car with > 100 ECUs and Industrial robots

Figure 1.1.3: Electronic appliances and Pacemaker

Use cases of real-time systems can be found in the following areas:

- Production
- Aerospace
- Automotive Electronics
- Medicine
- Military Technology
- e-Banking
- e-Trading
- Telecommunications
- Network Management
- Power Generation/Management
- Navigation

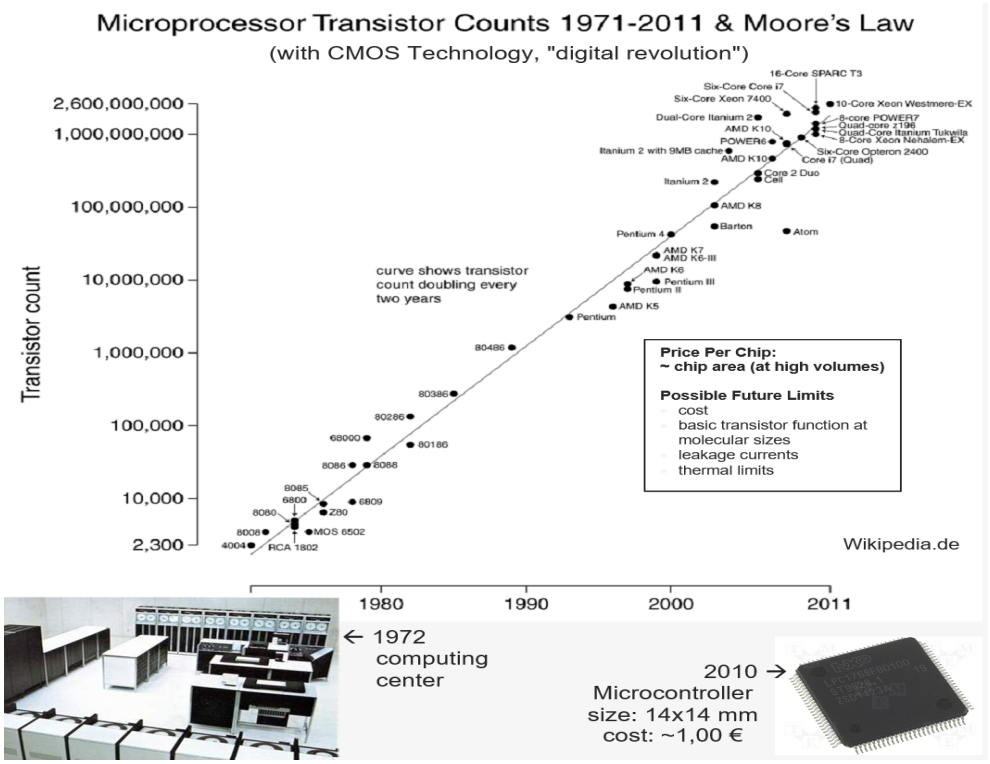


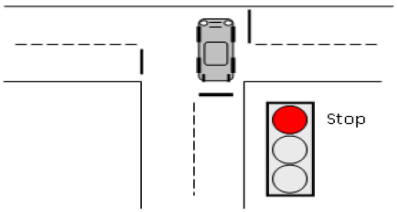
Figure 1.1.4: Moore's Law

.

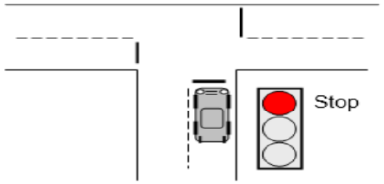
- Software for Real-Time Systems differs substantially from software for general purpose computers (PC) due to the **requirements on the real-time behaviour**!
- Many methods of **classical software development** of non-real-time systems can not be used due to the lack of predictability.
- For the development of real-time systems, special methods are required.

1.2 Requirements

The validity of an operation of a real-time system depends on it's logical result, and physical time this result is available



(a) logical correct, but timing incorrect

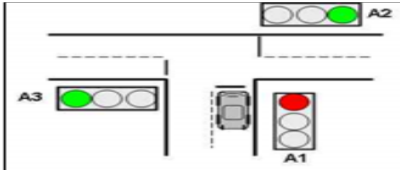


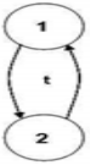
(b) logical correct, and timing correct

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

- Non Real-Time Systems
- Real-Time Systems

Example: Real-Time Requirements with a Traffic Light Control





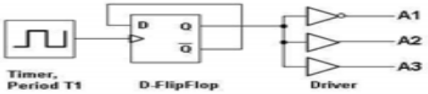
Problem Analysis

Finite State Machine (FSM)
Phase 1: 0 ≤ t < T1 → State 1
Phase 2, T1 ≤ t < 2·T1 → State 2

State 1: A1 = red A2 = A3 = green
State 2: A1 = green A2 = A3 = red

Realization in Hardware

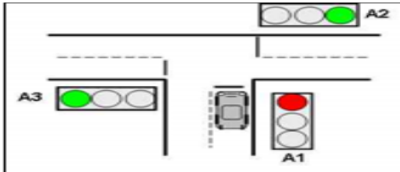
state encoding: 1:Q = 0, 2:Q = 1
output encoding: **red**: 1 **green**: 0

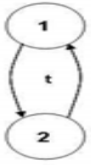


typ. Delay ~ 10 ns

Advantages
- Realization relatively cost effective with standard logic, PLA or FPGA
- Real-Time aspects are usually uncritical

Disadvantages
- Inflexible solution: Adaptations, more features e.g.: yellow-phases, synchronization, priority control,... involve hardware changes !
- Variants are relatively expensive





Problem Analysis

Finite State Machine (FSM)
Phase 1: 0 ≤ t < T1 → State 1
Phase 2, T1 ≤ t < 2·T1 → State 2

State 1: A1 = red A2 = A3 = green
State 2: A1 = green A2 = A3 = red

Realization in Software (FSM)
state encoding: 1:Q = 0, 2:Q = 1
output encoding: **red**: 1 **green**: 0

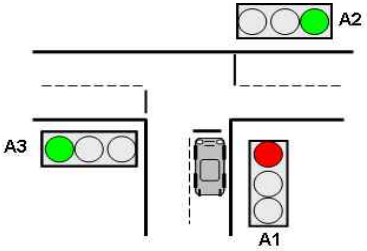
```
// called periodic with T1
void AmpelTask() {
    static int state = 0;
    state = !state; // state change
    if (state == 0) {
        A1 = 1;
        A2 = A3 = 0;
    } else {
        A1 = 0;
        A2 = A3 = 1;
    }
}
```

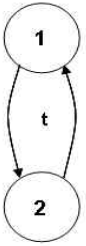
critical
Delays can appear !

Delay: 1 µs .. x ms !

Advantages
- Realization very cost effective with µC (Embedded System with/without network „**IoT**")
- Standard-Hardware + Embedded Software (C/C++)
- Flexible solutions: Adaptations, more features e.g.: yellow phases, synchronization, priority control,... by software-updates, without changing hardware !

Disadvantages
- Real-Time Requirements have to be fulfilled → ***Real-Time Programming***





Problemanalyse

Endlicher Zustandsautomat (FSM)
Ampelphase 1: 0 ≤ t < T1 → Zustand 1
Ampelphase 2, T1 ≤ t < 2·T1 → Zustand 2

Zustand 1: A1 = rot A2 = A3 = grün
Zustand 2: A1 = grün A2 = A3 = rot

Realisierung in Software (FSM)
Zustandscodierung: **1**:state = 0, **2**:state = 1
Ausgangscodierung: **rot**: 1 **grün**: 0

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

kritische Verzögerungen !

Delay: 1 µs .. x ms !

Vorteile
- Sehr kostengünstig mit µC realisierbar (Embedded System mit/ohne Vernetzung „**IoT**“)
- Standard-Hardware + Embedded Software (C/C++)
- Flexible Lösung: Anpassungen, weitere Features (Gelbphasen, Grüne Welle, Bedarfssteuerung,... ohne Hardwareänderungen per Software-Update)

Nachteile
- Echtzeitanforderungen müssen berücksichtigt werden → ***Echtzeitprogrammierung***

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

```
1  // called periodic with T1
2  void AmpelTask() {
3  static int state = 0;
4  state = !state; // state change
5    begin(mutex);
6  if (state == 0) {
```

```
7       A1 = 1;
8       A2 = A3 = 0;
9     } else {
10      A1 = 0;
11      A2 = A3 = 1;
12    }
13      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:

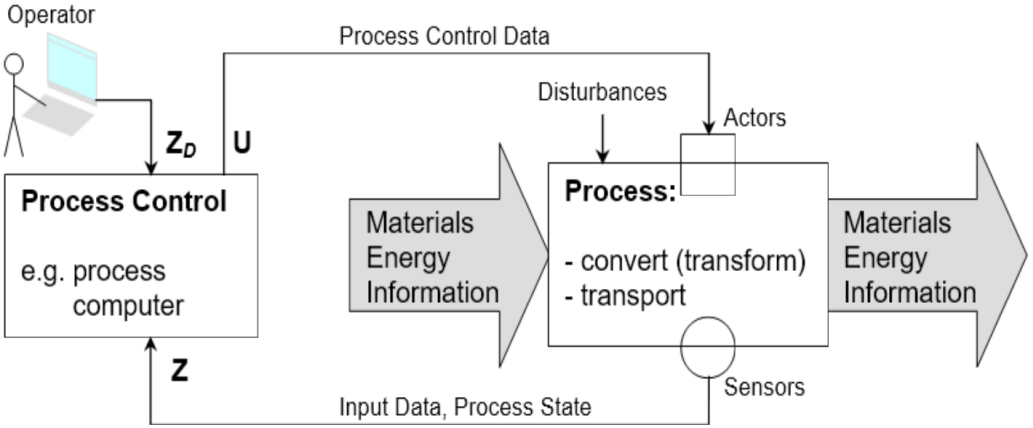
- In computer science, **real-time** computing (RTC), or reactive computing, is the study of hardware and software systems that are subject to a ”real-time constraint”, i.e., operational **deadlines** from **event to system response**.
- By contrast, a **non-real-time system** is one for which there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

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).

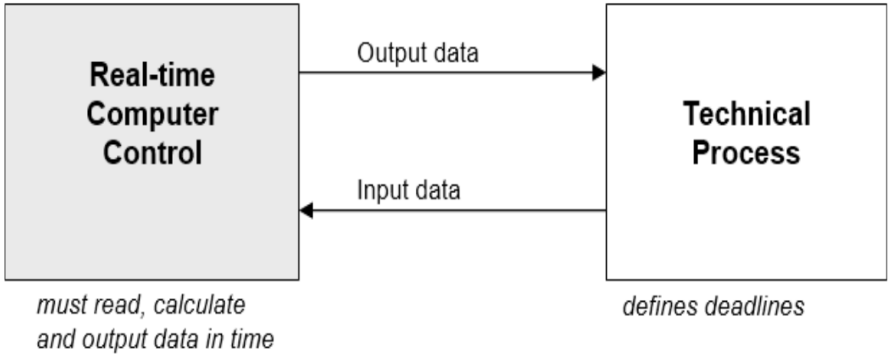
The delay time in the loop is of ***crucial importance*** for the ***stability*** of the ***entire system***!

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**.

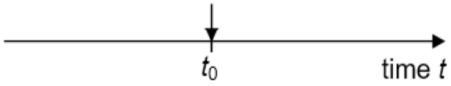
For a real-time system also means that it **monitors the temporal validity** of information and prevents the use of invalid data !

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

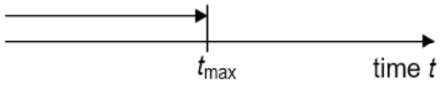
1. Precise Time, Exact Time



   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.

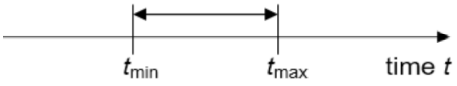
2. Latest Time Limit, Deadline



   A 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, slider, ... e.g. $t_{max} < 50$ms with human reaction times "*immediately*"

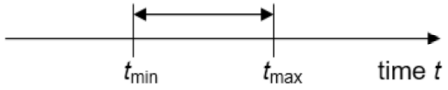
3. Earliest Time Limit



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 Interval



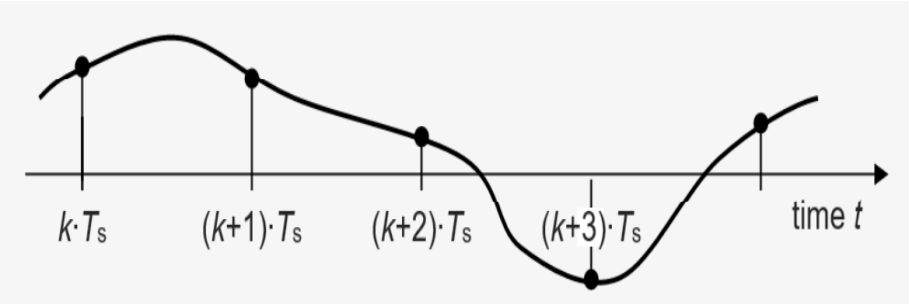
An action must be executed within a time interval [t_{min}, t_{max}], thus at any time earlier than t_{max} and later than t_{min}.

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]

The basic theorem of Shannon / Kotelnikov in signal theory states, that any analog signal x(t), which is sampled at constant frequency fs can be perfectly reconstructed from the sampling series x(k.Ts), apart from a constant delay, if the signal bandwidth Bx is not larger than half the sampling frequency fs = 1 / Ts, so Bx < fs / 2.

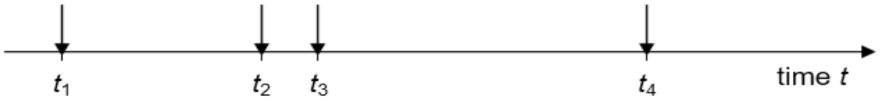
Correct sampling: tagesschau_44.1.wav
Sampling theorem violation: tagesschau_D10.wav

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.T_s
computation of a signal sample	time interval [k.T_s + T_{ADC}, (k+1).T_s]
output of the calculated signal sample	exact periodic sampling at (k+1).T_sy

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

- **sufficient processing speed** to process the input and output data at the required speed (sampling frequency), which the technical process requires
- **deterministic time behavior**

Hard and Soft Real-Time Systems

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: Aerospace flight controls, Airbag ignition, pacemakers.

2. **Soft Real-Time Conditions**

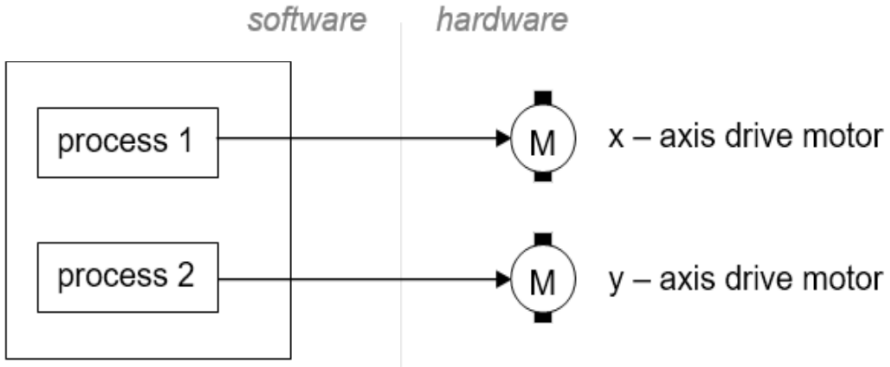
   → The deadlines must be met but with a degree of flexibility. The deadlines can contain varying levels of tolerance, average timing deadlines, and even statistical distribution of response times with different degrees of acceptability. A missed deadline does not result in system failure, but costs can 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, toasters, refrigerators.

Concurrency

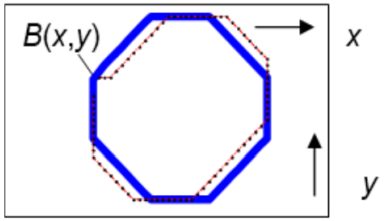
Real-time systems must generally treat multiple inputs and outputs simultaneously. Thus, the simultaneous addition to the timeliness of the second general requirement for real-time systems.

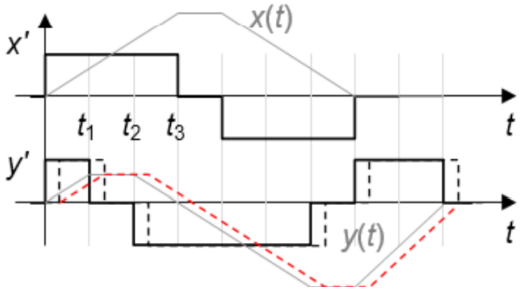
Example is about a CNC machine tool, where the x-y axes must be controlled simultaneously (synchronously, concurrently) to move the tool along a predetermined path $B(x, y) = B(x(t), y(t))$.



There is an relative time condition for both processes acting concurrently on the two

drive axes x and y:





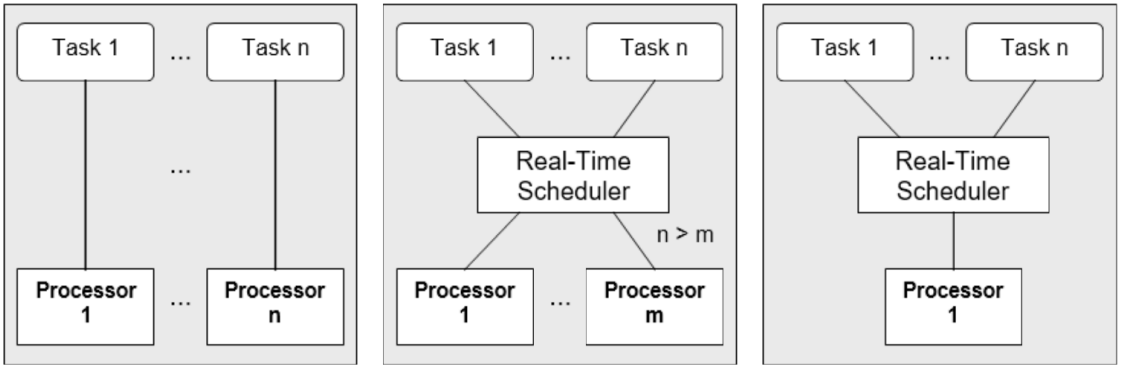
If the time axis of x and y are shifted by some amount of time ΔT (e.g. due to different clocks for x- and y-drive control), the resulting path $B(x, y) = B(x(t), y(t\text{-}\Delta T))$ will be wrong.

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**)



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).

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.

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

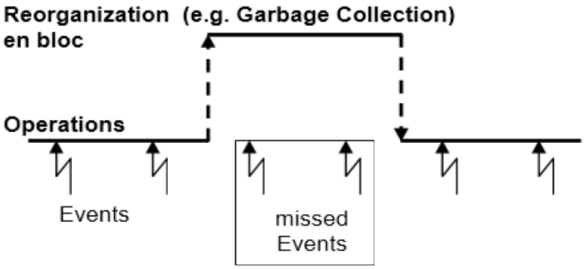
→ preferred realization of most embedded systems.

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

- Power Plant Controls
- Production Facilities
- Heating and Air Conditioning Systems
- Communication
- Medical Applications ...

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

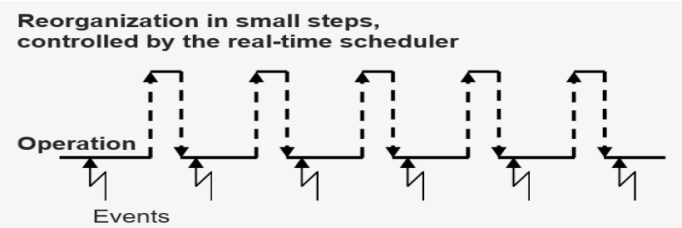
```
/* reorganization free algorithm
   for calculation of a factorial */

int fact(int n)
{
  int f = 1;
  for (i = 1; i < n; i++) {
    f *= i;
  }
  return f;
}
```

```
/* algorithm needs reorganization
   with calculation of a factorial */

int fact(int n)
{
  if (n > 1)
    return n*fact(n-1); // dynamic
  else                  // memory
    return 1;           // needed here
}
```

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:

```
/* dynamic memory management:
   not real-time capable due to
   stdlib: malloc/free */

int foo(int n)  // periodic call
{
  int *buf = (int*)malloc(n);
  int x = -1;
  if (buf) {
    x = some_algorithm(buf,n);
    free(buf);
  }
  return x;
}
```

```
/* static memory management:
   fully real-time capable */

int buf[NMAX];  // static alloc

int foo(int n)  // periodic call
{
  int x = -1;
  if (n <= NMAX) {
    x = some_algorithm(buf,n);
  }
  return x;
}
```

Not OK with Real-Time programming
(Exec.-Time of malloc() non deterministic)
- but: malloc() in initialization possible

OK with Real-Time programming !

1.3 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**).

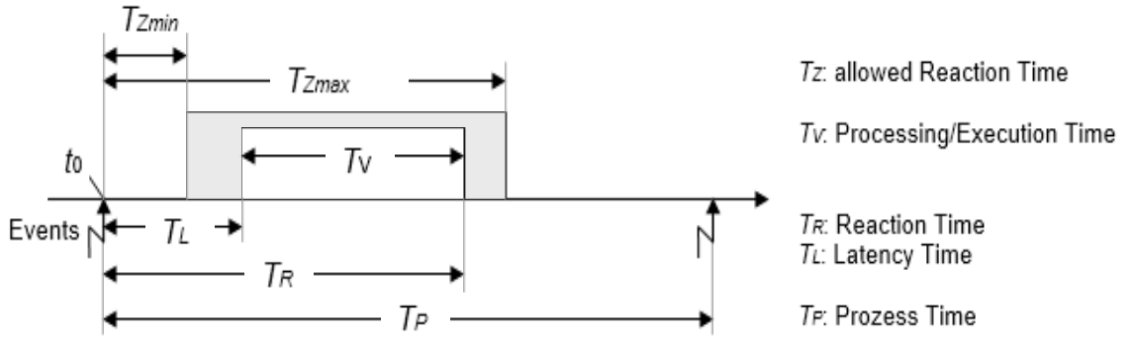
For the execution of a job i the computer needs **processing time** T_{Vi} (from de: *Verarbeitungszeit*) which is also called **execution time**.

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:

$$Tv = N/P \qquad [Tv] = ops/(ops/sec) = sec \tag{1.1}$$

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 1.1.



Timeliness, Response time and Maximum Response Time

Timeliness means, that a task required at the time t_0 is done **not before** a specified time t_0 + T_{Zmin} and is done **no later** than a specified time t_0 + T_{Zmax}

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

$$\rho = Tv/Tp \tag{1.2}$$

The total utilization $_{ges}$ is the sum of the utilization of all individual jobs 2.2. 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 ρ_{ges} is less than 100%:

$$\begin{aligned} \rho_{ges} &= \sum_{i=1}^{n} \rho_i = \sum_{i=1}^{n} \frac{T_{Vi}}{T_{Pi}} \\ \rho_{ges} &< 1 \end{aligned} \tag{1.3}$$

1.4 Real-Time Evidence

In principle, the real-time evidence is based on

1. Evaluate the relevant characteristics of the technical process,
   - number of different requirements
   - Minimum processing time for each request
   - Minimum allowable response time for each request T_{Zmin}
   - Maximum allowable response time for each request T_{Zmax}
   - dependencies between events
2. Identify the maximum processing time T_{VMax} (WCET = Worst Case Execution Time) for each request
3. Check the **load condition** (2.2)
4. 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).

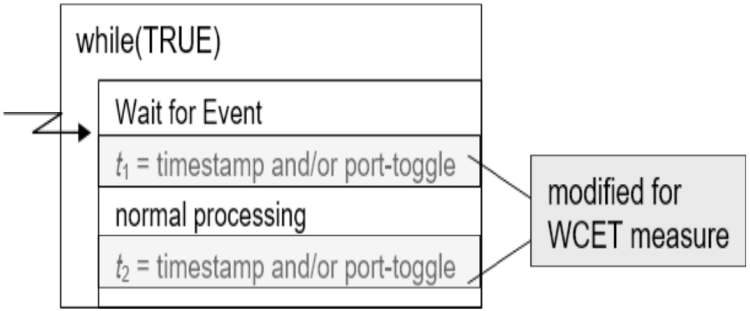
Therefore, the WCET at best be estimated. In principle, two methods for determining the WCET can be distinguished:

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

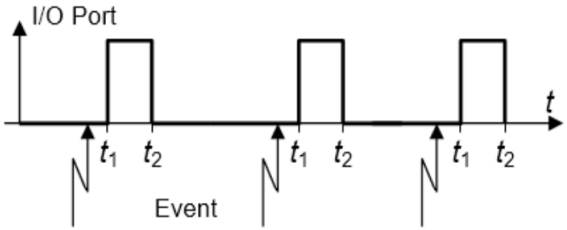
To measure this time can be divided into two basic methods:

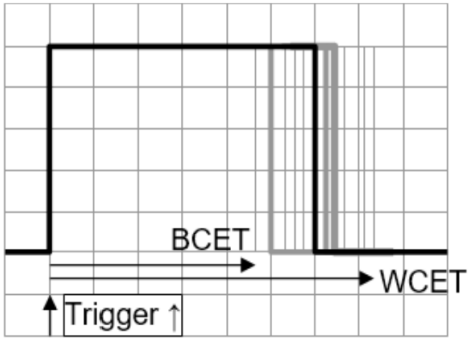
1. 1st Measurement by the measure out piece of code itself (see above).The piece of code will be amended such that every time an event arrives and when the reaction took place, a time stamp t_1 (free running timer) is stored. At the end of the code, a second time stamp t_2 is stored.
   $\rightarrow$ the difference T_{Vi} = t_2 - t_1 is the actual execution time.

   The 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:

(a) Independent of a programming language

(b) Relatively easy to implement

(c) Can run as autonomous self-diagnosis

Disadvantages of the method:

- The WCET of a code sequence can not be guaranteed, after all, it depends on many factors (background, loop iterations, caches, branches, etc ..).
- 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).
- 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.
- A test environment is required (which provides the input data to create).
- The code sequence to be measured out must be modified (for storing time stamps) → **timers**.
- 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 micro-controller).

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

$t_{Vmax} \leq T_{SA}$ Â· 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).

```
// ISR() is called with frequency
// FS * TSCNT
ISR(SIG_OVERFLOW0)
{
  // LED 1 blinks at PORTB
  if (cnt < (TSCNT>>1)) // TSCNT / 2
    PORTB |= 1;         // LED 1 on
  else
    PORTB &= ~0x01;     // LED 1 off

  cnt++;
  if (cnt == TSCNT) cnt = 0;
}
```

```
    if (cnt < (TSCNT>>1)) //
a4:   20 34          cpi   r18, 0x40   ; 64
a6:   10 f4          brcc  .+4         ; <__stack+0xd>
    portb |= 1;      // LED 1 on
a8:   91 60          ori   r25, 0x01   ; 1
aa:   01 c0          rjmp  .+2         ; <__stack+0xf>
   else
     portb &= ~0x01; // LED 1 off
ac:   9e 7f          andi  r25, 0xFE   ; 254
   cnt++;  if (cnt == TSCNT) cnt = 0;
c2:   82 2f          mov   r24, r18
c4:   8f 5f          subi  r24, 0xFF   ; 255
c6:   80 93 60 00    sts   0x0060, r24
ca:   80 38          cpi   r24, 0x80   ; 128
```

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.

$T_{Vmin} = \min(T_{Vi})$

2 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

- Task Management
- Resource Management
- Communication
- Synchronization
- Protection

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

- Respect for **timeliness** and **concurrency**
- Respect for **availability**

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

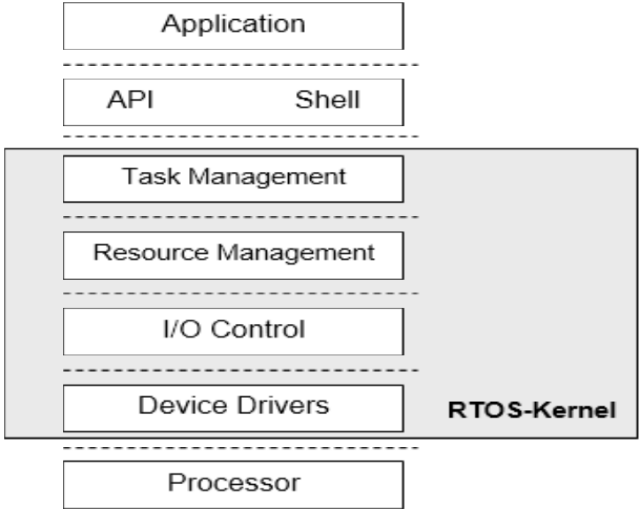
2.1 Structure of an RTOS

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.

- **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.



- **I/O (Input-/Output) Control**, realizes the logical, hardware-independent device control.
- **Resource Management:** responsible for (allocation) and de-allocation (release) of memory and I/O resources.
- **Task Management:** Responsible for the allocation of the processor to each task.
- **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**.

2.2 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.

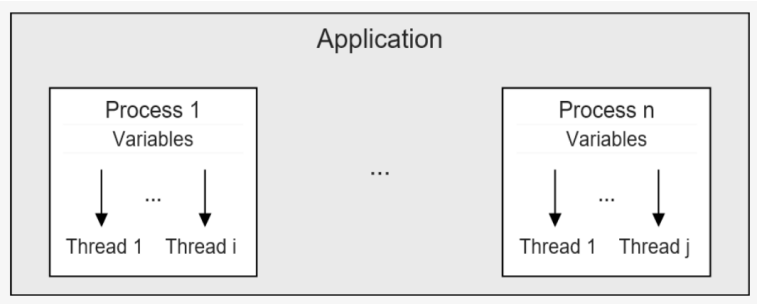
A **computational process** or **process** (also called **task**) is running as a computer program together with all the variables (including register states) and resources → each process has a **main()**

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()**



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.

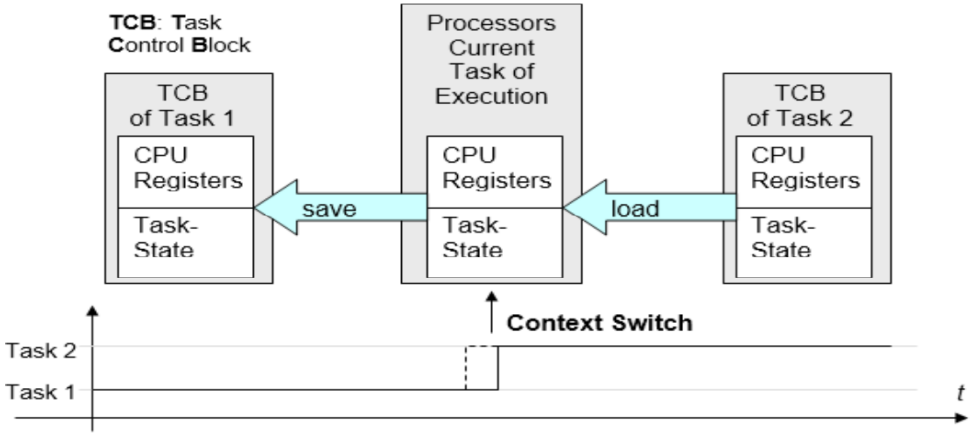
A **task** realized as a **thread** is called a **lightweight process** that exists within a single process. It uses the variables and resources of the process. All threads within a process **share the same address space**. Communication can take place over any global variable within the process.

- 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 fulfil the **real-time requirements**, efficiency is usually more important than protection. Many real-time applications use **threads within a single task**.

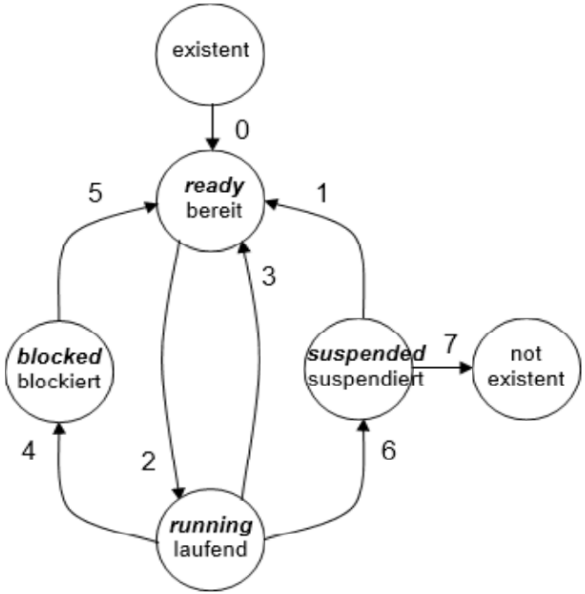
From the perspective of the real-time conditions (timeliness, concurrency, availability), **threads** aren't different from **processes**, both are usually identified by the term **task**.

Multitasking 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:



State Transients

0 Task is generated
1 Task resumes
2 Processor assigned
3 Task is displaced
(e.g. by preemption)
4 Resources missing
5 Resources available
6 Task suspended
7 Task eliminated

- **existent**
- **ready (**waiting**)**The task is ready, all conditions are fulfilled, resources are allocated, the task is waiting for the allocation of the processor.
- **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.
- **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).
- **suspended** (finished)A task is suspended from its normal operations by another task. It can be resumed later
- **not existent**

2.3 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.

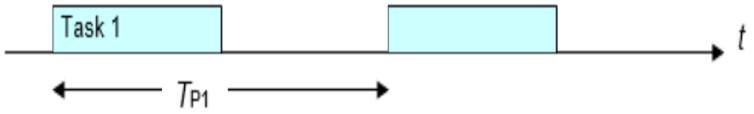
A **real-time scheduler** must divide all ready (runnable) tasks to the processor, so that all time conditions are met. The set of tasks managed by the real-time scheduler is called **taskset**.

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

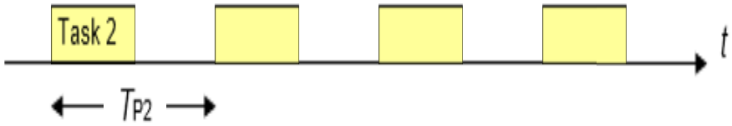
$$H = \frac{DemandedProcessorTime}{AvailableProcessorTime} \tag{2.1}$$

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 %

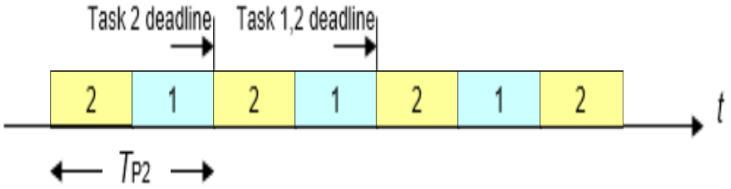


A second periodic task with a period of T_{P2} = 100 ms and an execution time of T_{e2} = 50 ms also causes a processor demand of H_2 = 50 %.

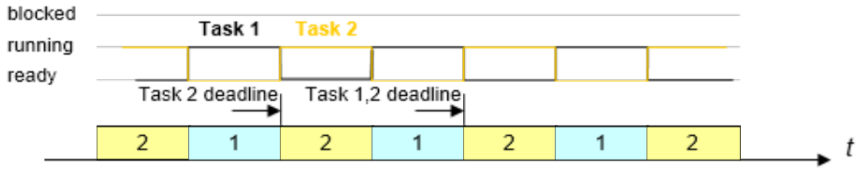


An implicit timing condition for each periodic task is: task execution must be finished before the next period starts (deadline) !

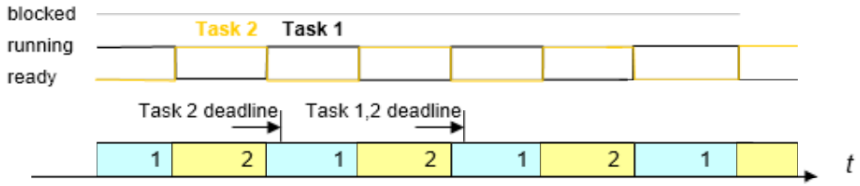
Both tasks cause a total processor demand $H = H_1 + H_2$ = 100 %.



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:

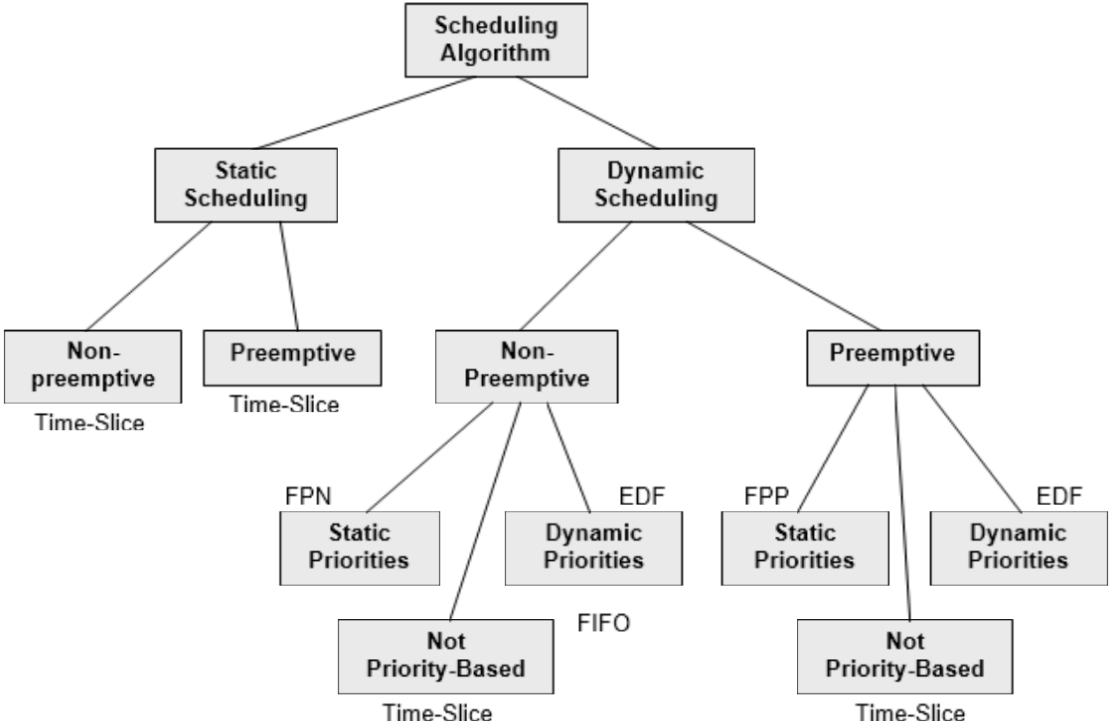


In general, for a tasklet of *n* periodic tasks, the total processor demand

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

T_{ei} execution time of task i, T_{pi} periodic time of task i

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

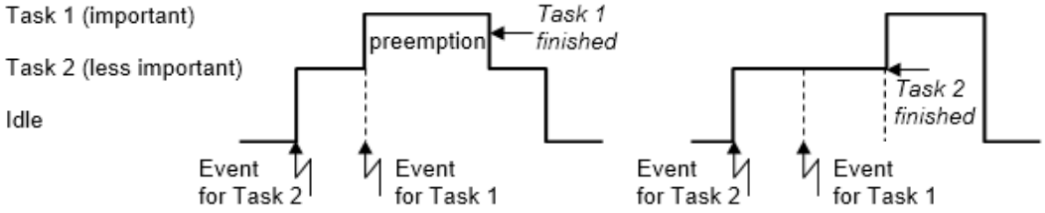
(Not to be confused with static/dynamic scheduling). The scheduler can use priorities for the allocation of resources (processor, memory, I/O,...) to a task. **Static priorities** are set before running the application and are never changed during runtime. **Dynamic priorities** can be adjusted for each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

Pre-emption

Prioritization is an important feature for a scheduler, which describes the capability for displacing a running task for later execution in favour of another task.

Pre-emptive scheduling

Pre-emptive 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**

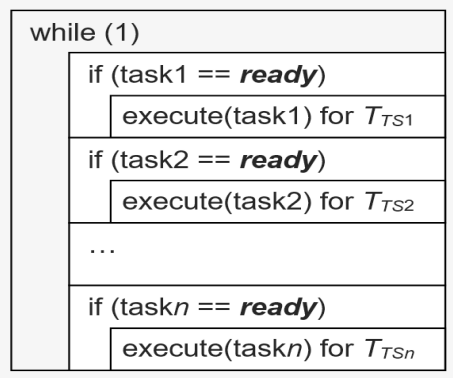
Non-preemptive scheduling

Non-preemptive scheduling or co-operative 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 pre-emptive 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:

All ***ready***-tasks are queued in a FIFO
Each task *i* gets assigned a time slice (time slice, quantum) T_{TSi}

- the task is preempted, i.e. displaced to the ***ready*** state
- the task is put at the end of the FIFO queue;
- The first task in the FIFO will be executed.

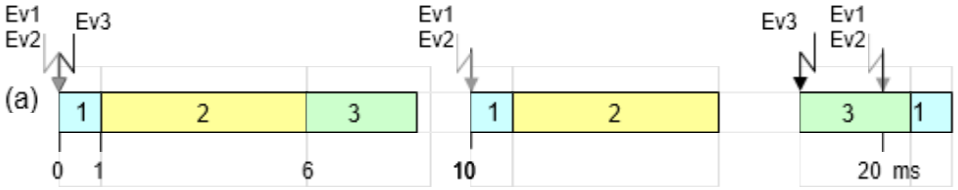
If a task changes state from ***blocked*** to ***ready***, it will be put at the end of the FIFO queue

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

By (2.2) 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 $\Sigma T_{TS} = 9$ ms:

Task T1: $T_{TS1} = 1 \,.T_{TS} = 1$ ms $\rightarrow H_1 = T_{TS1}/\Sigma T_{TS} = 1$ ms / 9 ms = 11 %
Task T2: $T_{TS2} = 5 \,.T_{TS} = 5$ ms $\rightarrow H_2 = T_{TS2}/\Sigma T_{TS} = 5$ ms / 9 ms = 55 %
Task T3: $T_{TS3} = 3 \,.T_{TS} = 3$ ms $\rightarrow H_3 = T_{TS3}/\Sigma T_{TS} = 3$ ms / 9 ms = 33 %

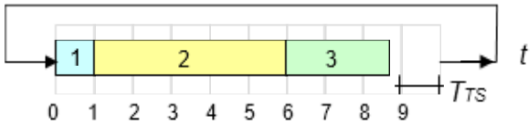


In this example the assigned time slices were chosen to be larger than the execution times of each task, in order to avoid preemption. With TSS a context-switch can occur at the end of a time-slice only. which can cause some *jitter* (as with task T1 above).

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 $N\hat{A}\cdot T_{TS}$, 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:

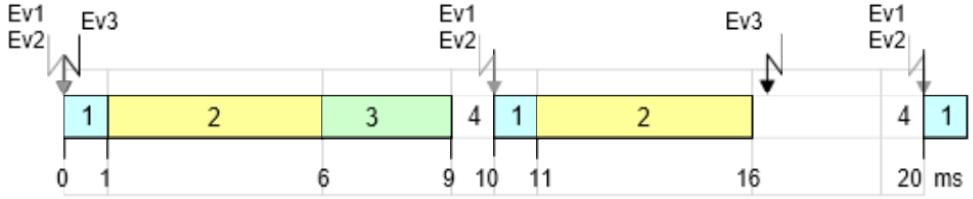


Example: FTS with $N = 10$

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: $T_{TS\,1}$ = 1 .T_{TS} = 1 ms → $H_1 = T_{TS\,1}/\Sigma T_{TS}$ = 1 ms / 10 ms = 10 % Task T2: $T_{TS\,2}$ = 5 .T_{TS} = 5 ms → $H_2 = T_{TS\,2}/\Sigma T_{TS}$ = 5 ms / 10 ms = 50 % Task T3: $T_{TS\,3}$ = 3 .T_{TS} = 3 ms → $H_3 = T_{TS\,3}/2\Sigma T_{TS}$ = 3 ms / 10 ms = 30 % Task T4 (idle) $T_{TS\,4}$ = 1 .T_{TS} = 1 ms → H_4 = 10 %

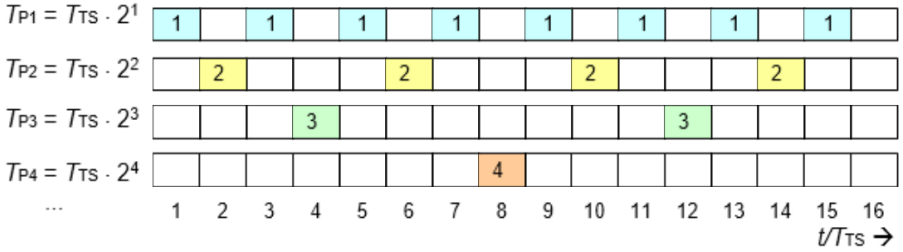


T3 runs for the second time at t = 26 ms the delay of 10 ms does not result in a T3-deadline violation. The idle task T4 provides a placeholder for unused CPU load, which can be used in modern CPUs for power saving features.

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

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

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 (2.3) 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

- quite suitable with multirate sampling systems (e.g. oversampling, when integer multiples of a common sampling frequency are needed).
- a very simple scheduling without priorities,
- suitable in a non-preemptive realization for even small microcontrollers without OS.
- Time and duration of execution for each task is guaranteed (apart from interrupts)

Disadvantages

- Binary time-slice scheduling is not fully flexible with arbitrary periodic times and execution times
- can turn into very complex software, especially with integrating tasks with execution time larger than T_{TS}.

Lab-Experiment AVR-Timer-Interrupts-(FTS-Scheduling)

```
void FTS_schedule()          // called periodically, with FTS_us
{
  if (FTSLICE(1, FTS_count)) {   // ---- 2^1 timeslice ----
    // --------------------------------
    TOGGLE(togpin);          // for WCET measurements by scope
    adc_in = analogRead(adcpin);
    TOGGLE(togpin);          // for WCET measurements by scope

  } else if (FTSLICE(8, FTS_count)) {  // ---- 2^8 timeslice ----
    // --------------------------------
    TOGGLE(outpinA);

  } else if (FTSLICE(9, FTS_count)) {  // ---- 2^9 timeslice ----
    // --------------------------------
    TOGGLE(outpinB);

  } else if (FTSLICE(10, FTS_count)) {  // ---- 2^10 timeslice ----
    // --------------------------------
    TOGGLE(ledpin);          // visual heartbeat ~ 1.0 Hz
  }
  FTS_count++;   // increment timeslice counter
}
```

Sketch for Arduino Nano (FixedTimeSlice.ino)

Example: 4 periodic tasks (on an AVR μC, from section Digital PID Control with 4 Tasks)

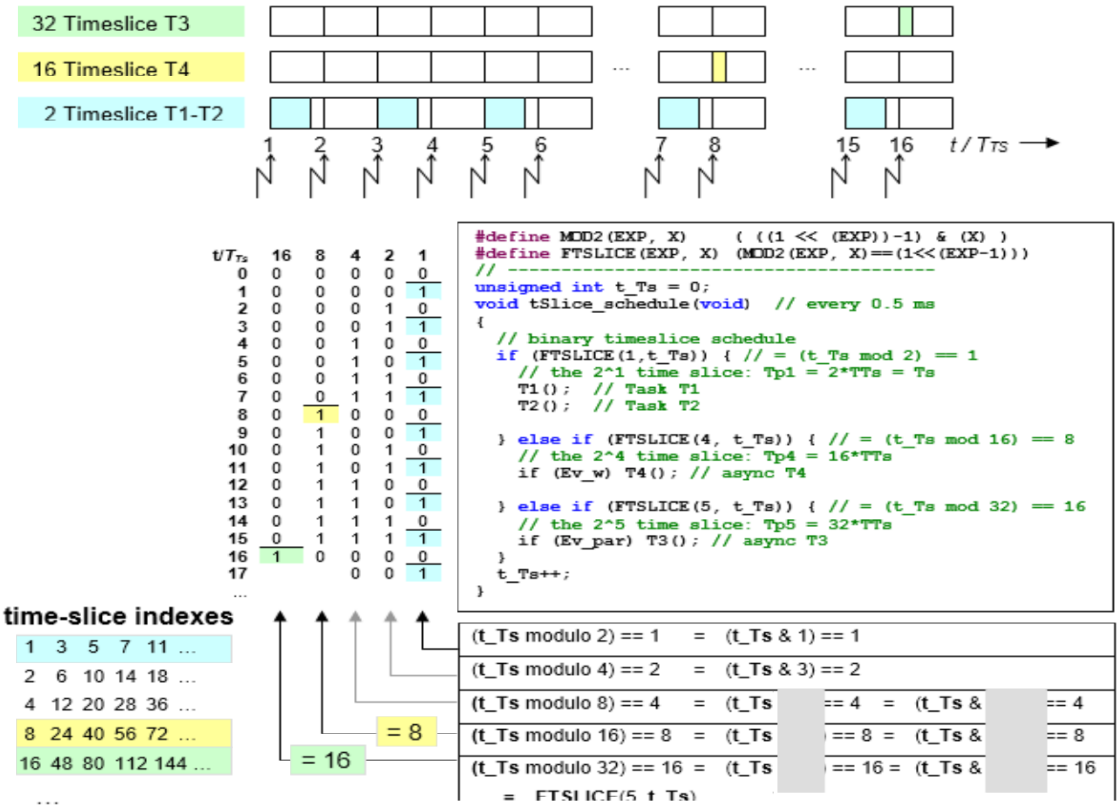
T_{p1} = T_{p2} = 1 ms, T_{p3} = 16 ms, T_{p4} = 8 ms

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 %, $\rightarrow$ 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:

$\rightarrow$ T1 und T2 executed successively in the 2 Timeslice 2Â·T_{TS} = 1 ms



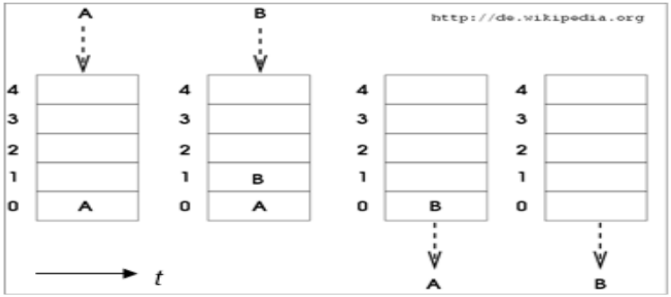
Proof for collision-free assignment of time-slice indexes (with $k,l \in \mathbb{Z}$ integers). For some period N we get time slices at indexes $n_k = k \bullet N + \frac{N}{2}$ e.g.: 2, 6, 10, 14, ..

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: Two tasks with FIFO scheduling

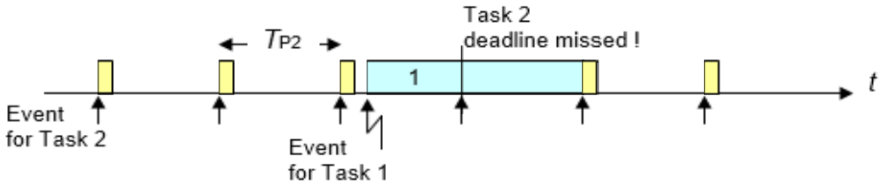
Task 1: T_{p1} = 150 ms, T_{e1} = 15 ms H_1 = 0.1

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 (2.2)

$H = H_1 + H_2 = 0.1 + 0.1 = 0.2$

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, pre-emptive 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, preempted) and the task with the higher priority is assigned to the processor immediately.
2. **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.

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

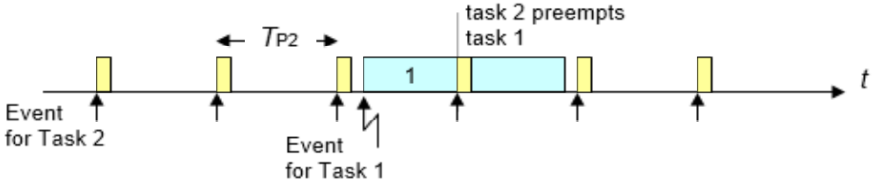
$$PR_i \sim \frac{1}{T_{pi}} \tag{2.4}$$

PR_i Priority task i, T_{pi} Periodic time of task i under the (rather idealized) conditions:

- Pre-emptive scheduling is used
- The periodic times T_{Pi} are constant
- The time limits (deadlines) are equal to the periodic times T_{Pi}
- The execution times are known and constant T_{ei}
- 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 (2.4)
Task 2: T_{p2} = 10 ms, T_{e2} = 1 ms higher priority by (2.4)



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

With periodic tasks with fixed priorities and deadlines equal to their periodic times, rate-monotonic scheduling (RMS) provides for an optimal priority allocation.

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)

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

n number of tasks

The limit (2.5) 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. Fur-

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

thermore, there are sometimes difficulties in approaching non-periodic processes by periodic processes with sufficient precision.

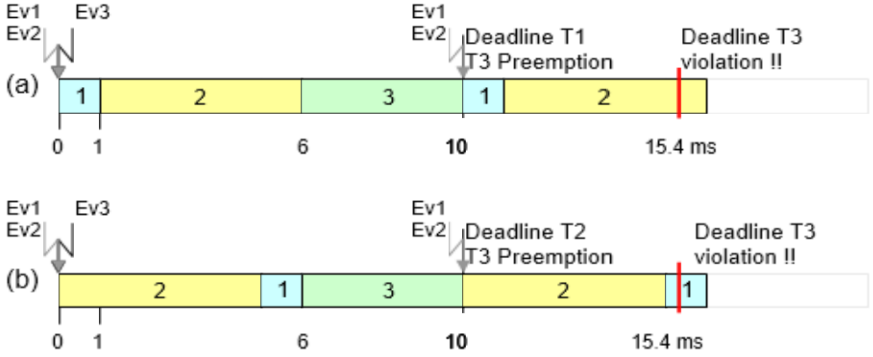
Example 1: 3 Tasks with $H_{max} > 0.778$, from [WörnB] :
Task T1: period T_{p1} = 10 ms, execution time T_{e1} = 1 ms $\rightarrow$ H_1 = 0.1
Task T2: period T_{p2} = 10 ms, execution time T_{e2} = 5 ms $\rightarrow$ H_2 = 0.5
Task T3: non-periodic, deadline = T_{d3} = 15.4 ms, execution time T_{e3} = 5.5 ms

By (2.2) the total CPU utilization H = 0.1 + 0.5 + 0.357 = 0.957, amost 100 % !But with (2.5) $H > H_{max} > 0.78$ deadlines will be violated with **FPP/RMS** with 3 tasks. FPP/RMS **can't deliver an executable schedule**, as this example shows:

By (2.4) 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 high priority
T2 medium priority	T1 medium priority
T3 low priority	T3 low priority

In both cases there is a violation of the T3 deadline, if all tasks get ***ready*** the same time at $t = 0$:



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.

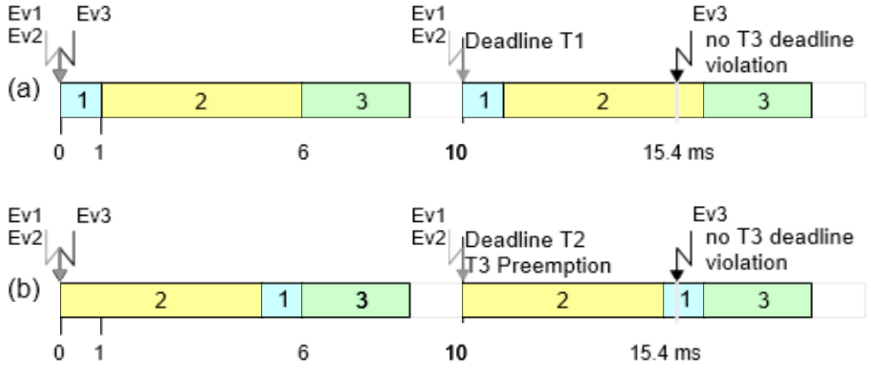
Example 2: 3 Tasks with $H_{max} < 0.778$, from [WörnB] :
Task T1: period T_{p1} = 10 ms, execution time T_{e1} = 1 ms $\rightarrow$ H_1 = 0.1
Task T2: period T_{p2} = 10 ms, execution time T_{e2} = 5 ms $\rightarrow$ H_2 = 0.5
Task T3: non-periodic, period T_{p3} = deadline T_{d3} = 15.4 ms, T_{e3} = 2.62 ms $\rightarrow$ H_3 = 0.17

By (2.2) the total CPU utilization H = 0.1 + 0.5 + 0.17 = 0.77 With (2.5) $H < H_{max}$ = 0.78 deadlines will be not violated with FPP/RMS with 3 tasks:

By (2.4) 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 high priority
T2 medium priority	T1 medium priority
T3 low 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:



→ 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)

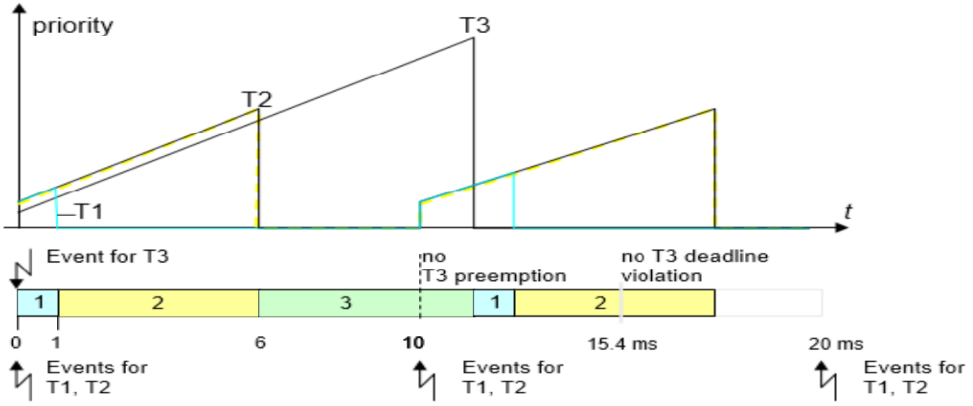
In Earliest-Deadline-First Scheduling (EDF) the Task Processor is granted to that ***ready***-task, which is closest to its time limit (deadline). This is achieved by assigning the task priorities according to the vicinity to the individual deadlines. EDF is dynamic scheduling with dynamic priorities, 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.

Pre-emptive 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 pre-emptive 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 = 0.1Task T2: period T_{p2} = 10 ms, execution time T_{e2} = 5 ms H_2 = 0.5Task 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 (2.2) 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

$$H<100\% for\ preemptive\ EDF\ scheduling \tag{2.6}$$

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.

2.4 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. Co-operation (sequence of access).

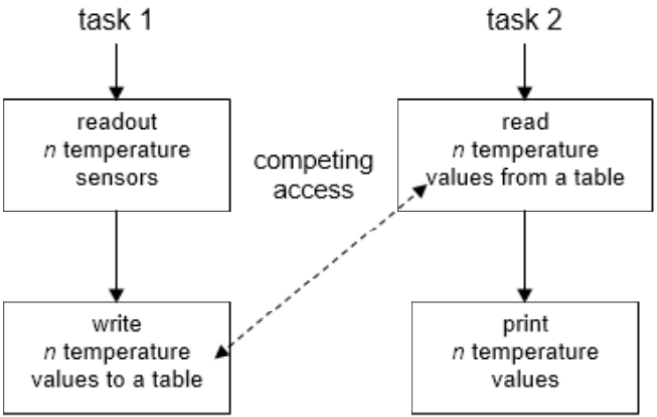
The problem of synchronization of tasks arises when these tasks are not independent from each other. Dependencies arise, when common resources are be used, then the access needs to be coordinated. Common resources may be

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.
2. **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).
3. **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:

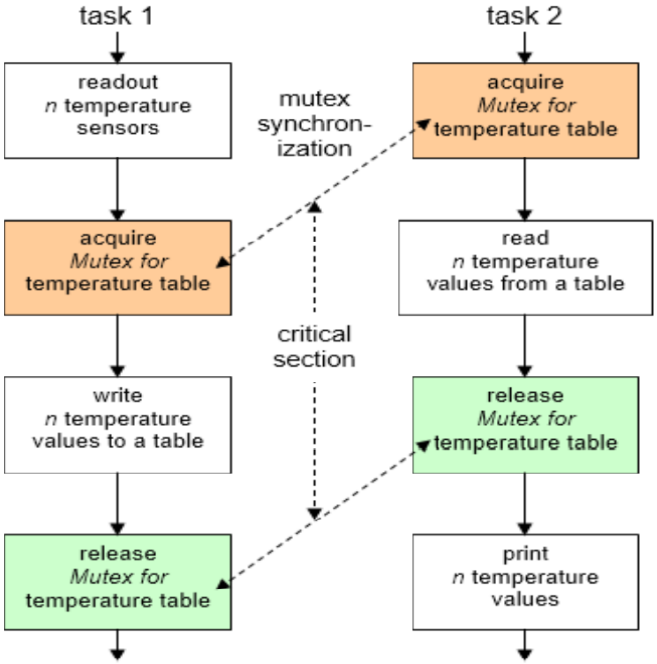
Task 1 is reading several temperature sensors, and stores these values in a table
Task 2 is reading this table, and prints out the temperature distribution.

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.

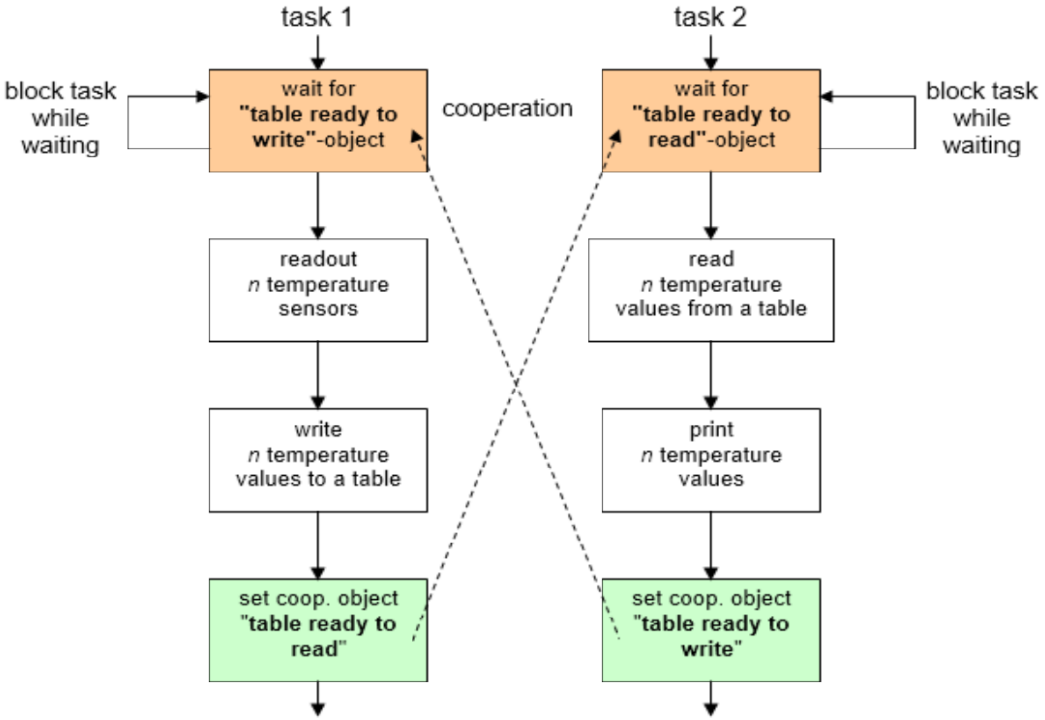


Example: of temperature measurement using the **mutex** synchronization.

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.

2. 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.


task 1
task 2
block task while waiting
wait for "table ready to write"-object
cooperation
wait for "table ready to read"-object
block task while waiting
readout n temperature sensors
read n temperature values from a table
write n temperature values to a table
print n temperature values
set coop. object "table ready to read"
set coop. object "table ready to write"

.

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**:

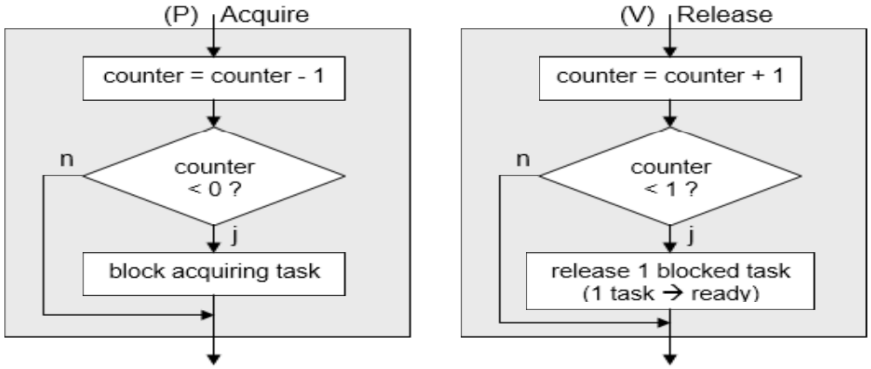
1. **Acquire** (also called ”**take**”, ”**signal**”, ”**lock**”, or ”**Passieren**”, (**P**))
2. **Release** (also called ”**give**”, ”**wait**”, ”**unlock**” or ”**Verlassen**”, (**V**))

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.



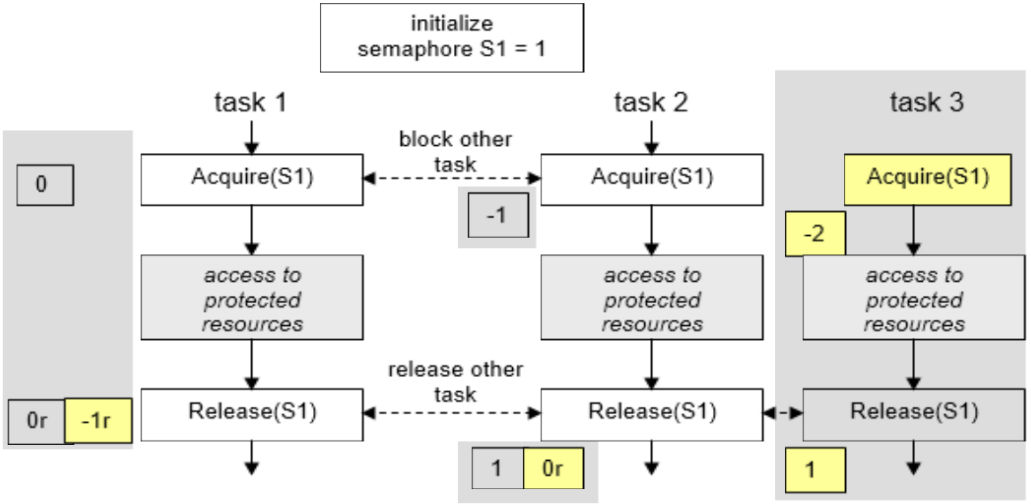
It is of crucial importance that the operations ”Passieren” and ”Verlassen” (as originally introduced by Dijkstra) are realized atomically, i.e. they cannot be interrupted by any other operation. Only then, a consistent handling of the counter variable is ensured. Semaphore objects are managed by the 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)$_{VXWorks}$

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

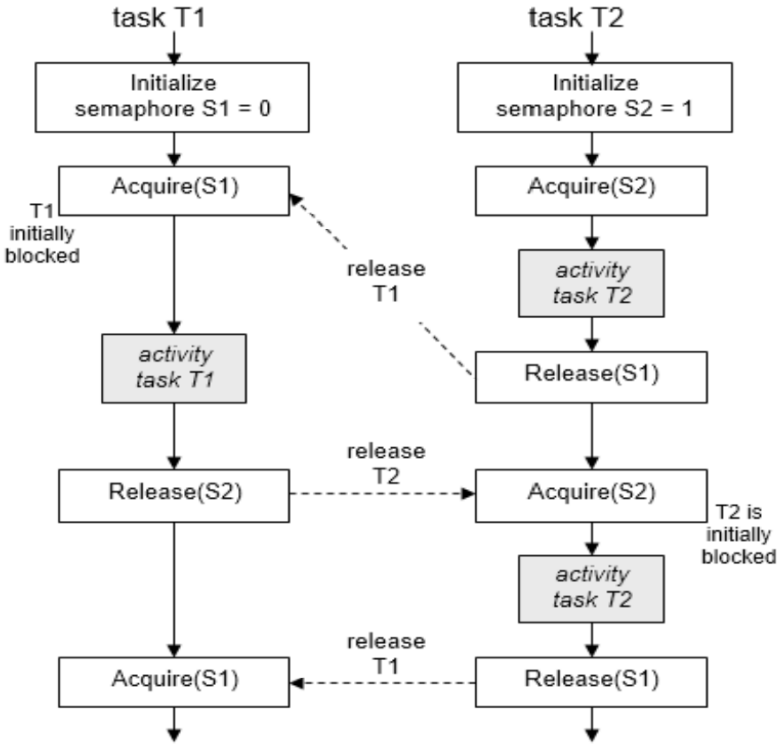


Example: Semaphore as Mutex in FreeRTOS

```
xSemaphore semAD;                    // a global Semaphore
int table[16];                       // a table of 16 temperatures

void vAppTask(void * pvParameters)
{
  semAD = vSemaphoreCreateBinary();   // Create a Semaphore, initially 1

  while(1) {
    ...
    xSemaphoreTake(semAD,portMAX_DELAY);    // <---- sync aquire mutex

    // read temperatures exclusicely ...
    Read_Temperatures(table);

    xSemaphoreGive(semAD,portMAX_DELAY);    // <---- sync release the mutex
  }
}

void vPrintTask(void * pvParameters)
{
  while(1) {
    ...
    xSemaphoreTake(semAD,portMAX_DELAY);  // <---- sync aquire mutex

    // print the table of temperatures exclusicely ...
```

```
26      Print_Temperatures(table);
27
28      xSemaphoreGive(semAD,portMAX_DELAY);  // <---- sync realease the mutex !
29    }
30  }
```

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

```
1 void init()
2 {
3   s1 =  vSemaphoreCreateBinary();     // Create a binary Semaphore
```

```
  s2 =  vSemaphoreCreateBinary();     // Create a binary Semaphore
  initSemaphore(s1,0);
  initSemaphore(s2,1);
}

void vTask1(void * pvParameters)
{
  while(1) {
    xSemaphoreTake(s1,portMAX_DELAY); // <---- sync take blocks task !
    t1_activity();            // main T1 activity
    xSemaphoreGive(s2,portMAX_DELAY); // <---- sync
  }
}

void vTask2(void * pvParameters)
{
  while(1) {
    xSemaphoreTake(s2,portMAX_DELAY); // <---- sync take blocks task !
    t2_activity();            // main T1 activity
    xSemaphoreGive(s1,portMAX_DELAY); // <---- sync
  }
}
```

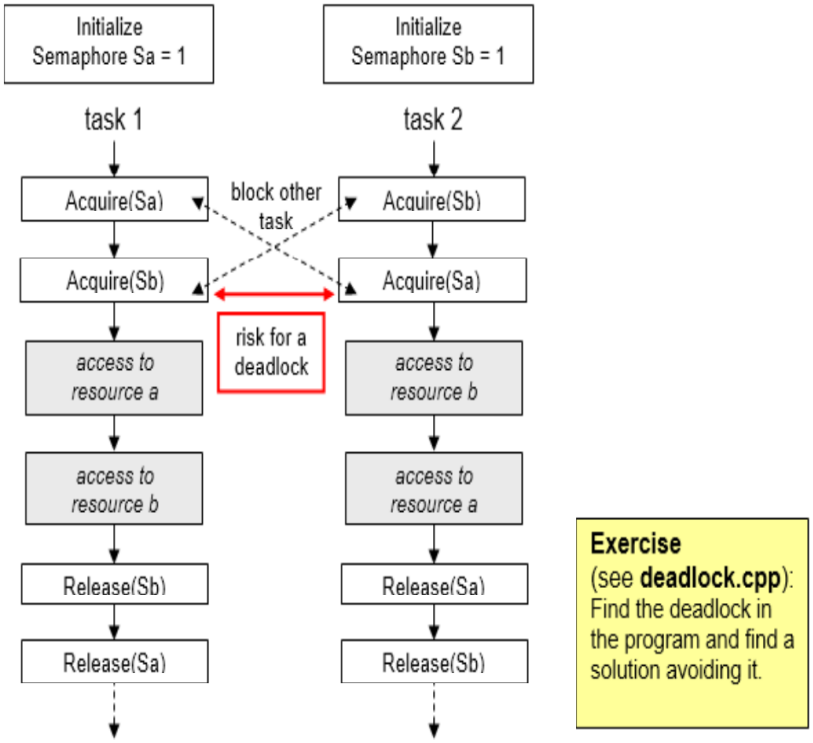
Deadlocks

Synchronization of tasks can lead to a deadlock (***block***, ***deadly embrace***), a situation, where the continuation of one or more tasks is blocked permanently. A simple example of a deadlock is an intersection with right-before-left right-of-way rule and four simultaneously arriving vehicles. After the priority rule any vehicle must wait for being right to another one, none of the Vehicles can drive. Only a break of the rule can release the deadlock.In task management, there are two types of deadlocks: deadlocks and live locks.

In a deadlock tasks waiting on the release of resources for which they block each other. This leads to a standstill of the waiting tasks. The previous example with the traffic intersection describes such a deadlock.

Deadlocks typically occur when multiple resources are to be protected at the same time over cross:

A simple rule to avoid deadlocks:



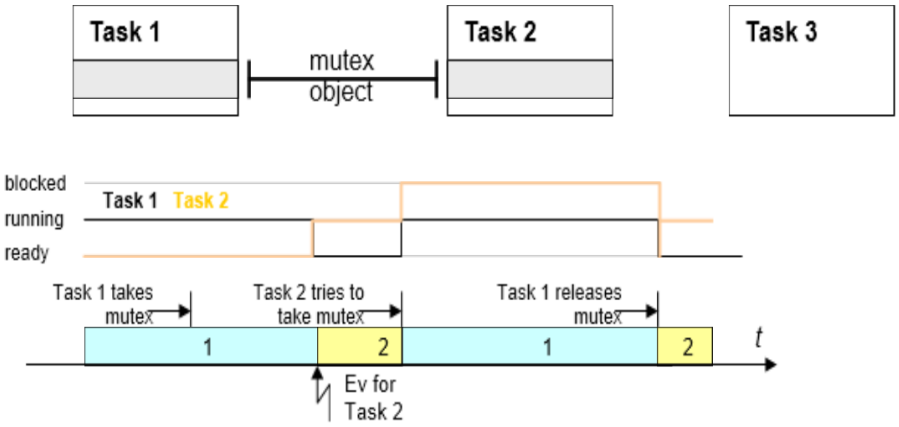
Exercise
(see **deadlock.cpp**):
Find the deadlock in the program and find a solution avoiding it.

If several resources are to be protected at the same time, all accessing tasks must acquire / release semaphores in the same order to avoid deadlocks.

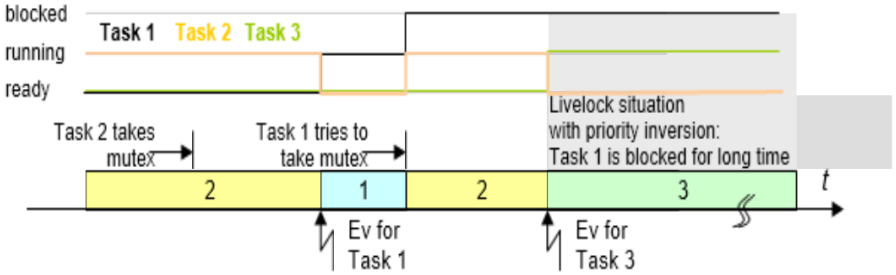
Livelocks

Livelocks (starvation) designate a condition in which a task is constantly inhibited from running by the conspiracy of other tasks. For example, when using a fixed lower-priority task never can get the processor due to the constant activity of higher-priority tasks.

Example FPP, task 1**: high priority**, task 2: **low priority**, task 3: **medium priority**



However, this can also happen in high-priority tasks, provided that a ***priority inversion*** occurs: the high-priority task 1 is livelocked by the lower priority tasks 2 and 3:



The problem of livelock by priority inversion can be solved by the technique of **priority inheritance** [WörnB] in the example above: task 2 inherits the high priority of task 1 (both wait for the same mutex), such that task 2 running in the critical section cannot be interrupted by a medium priority task 3.

Task Communication

Synchronization and communication tasks are very closely related. **Synchronization** can be thought of as **communication without information**. On the other hand, communication can be used for synchronization.

There are two basic variants of the task communication:

1. **Shared memory:**
   The data exchange is via a shared memory.
   The synchronization at what time read or write access occurred is done by a semaphore.
2. **Messages (Events):** The data exchange and synchronization is done via sending messages.

In general, the communication via shared memory is faster than communication via messages. Therefore, this variant is preferred used in real-time operating systems. In spatially ***distributed systems***, this is not possible because there is no physically shared memory. Here communication must take place via messages ($\rightarrow$ message queues).

Events

With a mutex synchronization the access of competing tasks to a jointly used resource was enabled. In contrast, for cooperating tasks time (and order) of execution has to be synchronized. This can be done by semaphores, but also with a simpler concept, an ***event***.

A task can wait on an ***event*** (event wait), which is set by another task. Once set, the waiting tasks gets released from its blocking state.

Example:
Each of 3 tasks (task A, task B, task C) acquires a measurement value, which shall be processed by a 4^{th} task W. If A, B or C has acquired a measurement value, this is written to a data memory (”datenspeicher”), which can store exactly one value. Task W retrieves its input from that data memory.

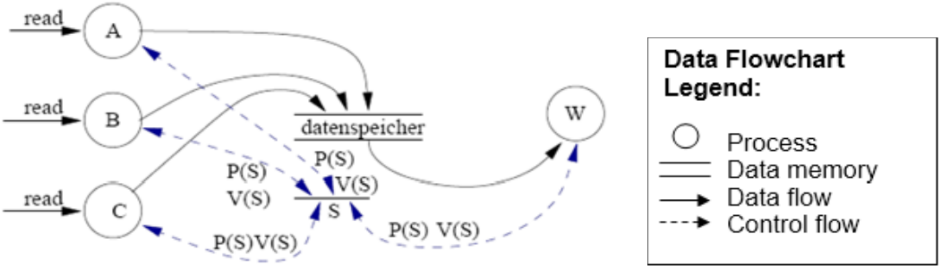


Figure 2.4.1: Data flow of an acquisition of measurement values: synchronization without events

Since the data memory is existing only once, it is a shared resource, and access to it is a critical section, which can be protected by a semaphore S

```
// Task A,B und C
  ...
  while( TRUE ) {
    read(kanal_A, buffer, sizeof(buffer));
    P(S); // enter critical section
    write( datenspeicher, buffer, sizeof(buffer));
   V(S); // leave critical section
}
...
```

```
// Task W
while( TRUE ) {
    P(S); // enter critical section
   read(datenspeicher, buffer, sizeof(buffer));
    V(S); // leave critical section
   WorkOnData( buffer );
}
```

Problems with synchronization without events:

> **Data memory** can be overwritten, without being read before by W
>
> W can read the **data memory** repeatedly.

Therefore, it is not sufficient to protect the shared resources by using a semaphore, yet another synchronization means is necessary: an ***event***.

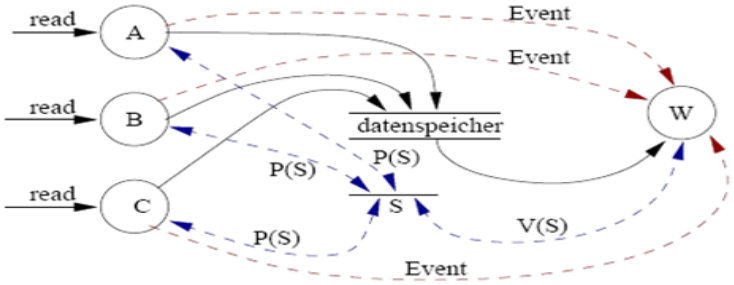
Definition:
An **event** is a synchronization object, which allows computing processes to give free the processor (by going to the **block**-state), up to a certain condition is met.

Basic operations of the **event** are:

Wait for an event, and

Send an event.

An **event** can **get "lost"**, if an event is sent when no process waits on it !
Since events are not stored -unlike semaphores-, events are used often in combination with a semaphore.



Data flow of an acquisition of measurement values: synchronization with events

In order to grant access to the data for the tasks A, B or C only, after task W has emptied the data memory, the operation ”enter critical section” and ”leave critical section” is distributed to different tasks, similar to cooperation synchronization with semaphores.

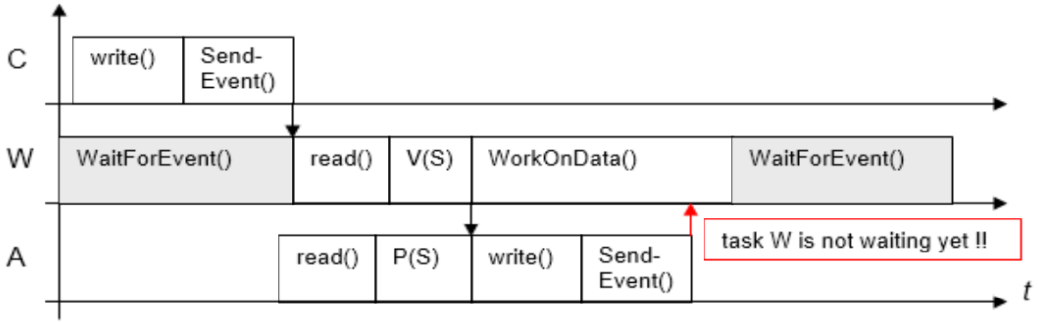
Each of the tasks A, B and C enters the critical section (by calling the ”P(S)”-operation), but never ”leaves the critical section” by itself, this release is now done by task W (by calling the ”V(S)”-operation). Task W must now be synchronized, so that it fetches the data from memory only, if these are available. Thus, Task A, B or C send an event on which task W is waiting:

```
// Task A,B und C
...
read(kanal_A, buffer, sizeof(buffer) );
P(S); // enter critical section
write( datenspeicher, buffer, sizeof(buffer) );
SendEvent( Task_W );
...
```

```
// Task W
while (1) {
  WaitForEvent( Task_W );
  read( datenspeicher, buffer, sizeof(buffer) );
  V(S); // leave critical section after read
  WorkOnData( buffer );
}
```

The critical section is thus defined from the time which an acquiring task A,B, or C writes a value into the data memory, until task W has read the value out of the memory.

Problem: Events can get lost, if sent when no task is waiting



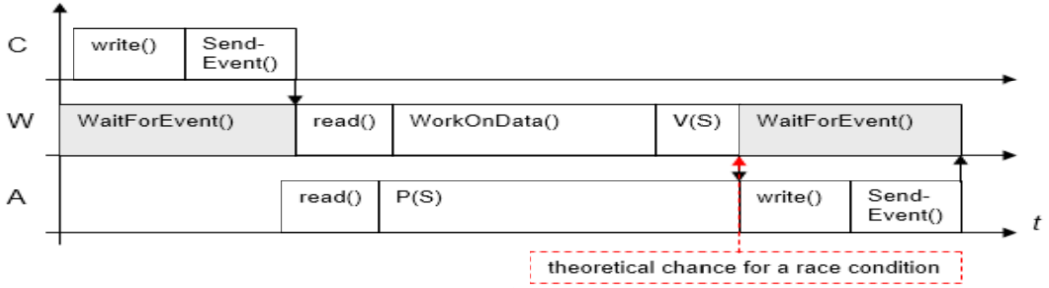
- task C causes task W to get running, starting with read()
- During read() in task W, task A has read new data
- task A is blocked by P(S), before W could release the semaphore
- task W releases V(S)
- task A continues with write() and SendEvent()
- the event gets lost, cause task W is busy WorkingOnData(), not waiting: ***critical race condition !!***

- data loss A.

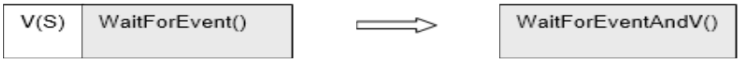
Solution: Move he WorkOnData() call into the critical section:

```
// Task W
while (1) {
  WaitForEvent( Task_W );
  read( datenspeicher, buffer, sizeof(buffer) );
  WorkOnData( buffer );
  V(S); // leave critical section after work
}
```

Now, with this solution, the conflict is solved, since immediately after the critical section is left, task W can wait for events there is no more critical race condition:



Since there is a theoretical (although very small) chance, **Posix** defines an atomar, non-interruptable operation combining the semaphore release and waiting on an event:




```
// Task W
P(S); // since WaitForEventAndV releases a semaphore
      // this most be locked (acquired) first
while(1) {
  read( datenspeicher, buffer, sizeof(buffer) );
  WorkOnData( buffer );
  // V(S); leave critical section after work, done above
  WaitForEventAndV(S); // Release semaphore and wait
}
```

```
// Task A,B und C
...
read(kanal_A, buffer, sizeof(buffer) );
P(S); // enter critical section
write( datenspeicher, buffer, sizeof(buffer) );
SendEvent( Task_W );
...
```

Signals

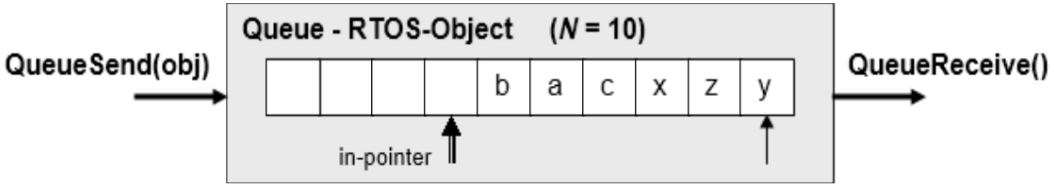
Signal are very similar to events. Signals can be processed asynchronously to the program (call of a signal-handler routine), with events the processing is synchronous (**wait_for_event**() - functions as part of the program).

- Signals are often used in UNIX programs.
- Signals they can be seen as software interrupts at the application level.

Messages Queues, Mailboxes

Message queues provide an asynchronous communication protocol, which means that the sender and receiver of the message must not interact simultaneously. Messages will be placed on a queue, where they are stored until the recipient retrieves one or more by a FIFO mechanism. The maximum amount of data a single message is usually limited.

A message queue without a restriction for the number of messages is called **mailbox**.



Example: FreeRTOS - Queue (www.freertos.org)

```
#include <FreeRTOS/queue.h>
...
struct AMessage
{
  portCHAR ucMessageID;
  portCHAR ucData[ 20 ];
}

xQueueHandle xQueue1;

void vATask(void *pvParameters)
{
  // Create a queue capable of containing 10 pointers to AMessage structures.
  // These should be passed by pointer as they contain a lot of data.

  xQueue1 = xQueueCreate( 10, sizeof( struct AMessage * ) );
}

...

// -------------------------  send from any task ----------------------------}
{
  // Send a pointer to a struct AMessage object.  Don't block if the
    // queue is already full.
  struct AMessage *pxMessage;

```

```
  pxMessage = \& xMessage;
  xQueueSend( xQueue1, (void *) \&pxMessage, (portTickType) 0 );
}
...

// ------------------------- receive with any task -------------------------}

{
  // Receive a message on the created queue.  Block for 10 ticks if a
  // message is not immediately available.
  struct AMessage *pxMessage;
  if( xQueueReceive(xQueue1, \&(pxRxedMessage), (portTickType ) 10 ))
  // pcRxedMessage now points to the struct AMessage variable posted

}
```

The **QueueReceive()** command places the calling task into the blocked (waiting) state if the message queue is empty. If not, it returns the object to which the read (out) pointer points to, and the read (out) pointer set to the next message object.

The **QueueSend()** command places the calling task into the blocked (waiting) state only,

if the message queue is full. If not, the object is being stored at the position the write (in) pointer references, and the pointer is set to the next free space in which the next transmitted (posted) message object is stored.

In many RTOS, a maximum blocking time can be specified (as with FreeRTOS). Lab-experiment ”Message Queue with ARM LPC4357 μC (FreeRTOS)”

Socket Interface

The most important interface for inter-process communication on different computers connected via Ethernet, is the socket interface.

By using sockets, data can be exchanged between processes, which are located on different hosts (distributed system).

The socket interface provides access to TCP/IP (connection-oriented) and UDP/IP (connectionless) protocols for communications.

Once a socket connection between two processes has been established, the data can be exchanged with some basic system functions (open, read, write, close ...), the so called Sockets Service Primitives.

RTP ProgC RepetitionWorkspace: ”socket client, socket server”

Primitive	Meaning	
SOCKET	Create a new communication end point	
BIND	Attach a local address to a socket	
LISTEN	Announce willingness to accept connections; give queue size	*blocking*
ACCEPT	Block the caller until a connection attempt arrives	
CONNECT	Actively attempt to establish a connection	
SEND	Send some data over the connection	
RECEIVE	Receive some data from the connection	*blocking*
CLOSE	Release the connection	

Python Sockets Programming

The server connects to a client and echoes its message:

```
server:> socket server.py
Connected by ('192.168.0.3', 4118)

server:>
```

```
client:> socket client.py
Received 'Hello World'

client:>
```

(A) Server script

First, the server executes LISTEN (line 12), and it remains blocked until a client connects. Then a client task executes CONNECT to establish a connection with the server (line 9). It must specify an IP address (the server's) and port number (must me common to both).

```
1  # http://docs.python.org/library/socket.html#socket-example
2
3  import socket
4
5  # Echo server program
6  import socket
7
8  HOST = ''                 # Symbolic name meaning all available interfaces
9  PORT = 50007              # Arbitrary non-privileged port
10 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11 s.bind((HOST, PORT))
12 s.listen(1)
13 conn, addr = s.accept()
14 print 'Connected by', addr
15 while 1:
16     data = conn.recv(1024)
17     if not data: break
18     conn.send(data)
19 conn.close()
```

(B) Client script

The OS then sends an IP packet to the opposite computer (peer). The client process is blocked until the server responds. If a confirmation is received, the client and server

```
# http://docs.python.org/library/socket.html#socket-example

# Echo client program
import socket

HOST = '192.168.0.10'     # The remote host
PORT = 50007              # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)
```

have established a connection and the connect() - or listen() function are released from blocking, and data can be exchanged until the end of the connection (close()).

2.5 Memory Management

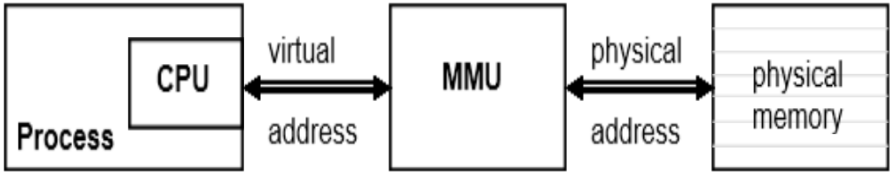
The main task of memory management is essentially providing access to the memory resource consisting of a hierarchy of cache, main memory (RAM), peripheral memory or NVRAM (non-volatile RAM, such as EEPROM) by

- the allocation of memory to the tasks,
- the coordination of accesses to shared memory areas,
- the protection of the storage areas of individual tasks
- the displacement of memory areas

One can distinguish between different forms of memory allocation:

1. **Static memory allocation:**
   The allocation of memory is made to a task before it is put into the ***ready*** state. The memory allocation does not change at runtime.
2. **Dynamic memory allocation:**
   The allocation of memory to a task is done at runtime and may change at any time.
3. **Non-displacing memory allocation:**
   Allocated memory must not be withdrawn from a task at runtime.
4. **Displacing memory allocation:**
   Allocated memory may be removed from a task at runtime, the memory is paged to peripheral memory (e.g. a swap-file).

The usage of an **MMU** (**M**emory **M**anagement **U**nit), which is quite common with many CPUs (x86, ARM Cortex A Series, ..), allows processors to deal with a virtual address space. Thus, memory access from the process is strictly separated from the OS and from other processes (= safety). However, with translation of the virtual addresses into physikalical addresses long delays can occur, which cannot be determined in advance! This is a contradiction to the required determinism with hard real-time requirements ! Thus, microcontrollers for applications having hard real-time requirements often do **NOT** have an MMU and use direct physical addressing instead (AVR, ARM Cortex M Series, ..)



2.6 Input/Output Management (I/O)

Along with task and memory management the I/O management is the 3^{rd} important component of a real-time operating system, especially with embedded systems, where I/O tasks for sensors and actors have hard real-time requirements.

In addition to that, synchronous an asynchronous serial interfaces, and networking sub-systems require high performance I/O-management.

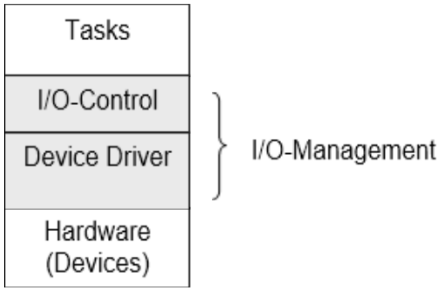
I/O Basics

The tasks of the I/O-management can be divided into two groups:

- Assigning and sharing of devices for the tasks
- Use of assigned and shared devices by the tasks

The connected devices differ greatly in speed and data format. The OS has therefore the task, to abstract from implementation details and provide a **simple, uniform and transparent interface**.

This can be achieved by a multi-layer architecture of the I/O management:



The main 2 layers of I/O-management:

1. **I/O-Control:**
2. **Device Driver:**

Example: I/O-management of a hard disk drive:

Typical interface functions provided by the I/O control layer

Function	Description
Create	Creates a virtual instance of an I/O device
Destroy	Deletes a virtual instance of an I/O device
Open	Prepares an I/O device for use
Close	Communicates to the device that its services are no longer required, which typically initiates device-specific cleanup operations
Read	Reads data from an I/O device
Write	Writes data into an I/O device
Ioctl	Issues control commands to the I/O device (I/O control)

The main functions of the I/O control layer are:

1. **Symbolic Names:** devices addressing by symbolic addresses, or names.
2. **Handling of I/O-Requests:** queuing, buffering I/O-requests queues.

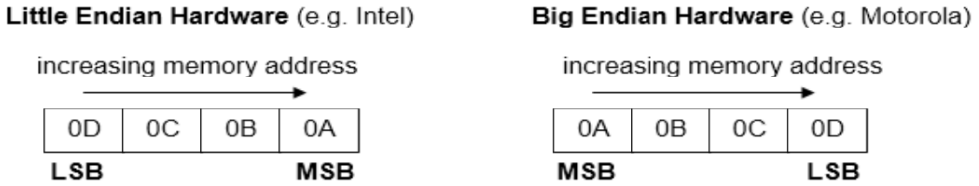
3. **Assignment of Devices:** assignment of the device (shared resource) to tasks dynamically or statically,using priorities for competing requests.
4. **Synchronization:** I/O-requests affect a tasks state, (e.g. waiting ***blocked*** state), interface between I/O-control to task-management.
5. **Device Protection:** prevent unauthorized access by unauthorized tasks - protection against deadlocks and livelocks due to competing access- resets after programmable timeouts (protect against software failure).
6. **Communication with the device drivers:** forwarded executable device requests to the device driver, handle feedback, e.g., status or error codes
7. **Buffering:** synchronization of different speeds of task-management and I/O-devices by temporary storage in buffer memory, handle buffter state events to the task/device
8. **Unique Data Format:** abstract from the device's physical data format, e.g. **endianness**.

The boundary between I/O control and device drivers is floating and system dependent. For efficiency reasons with some RTOSes, functionality is moved into the device driver,

which increases the effort for adapting new devices.

Endianness

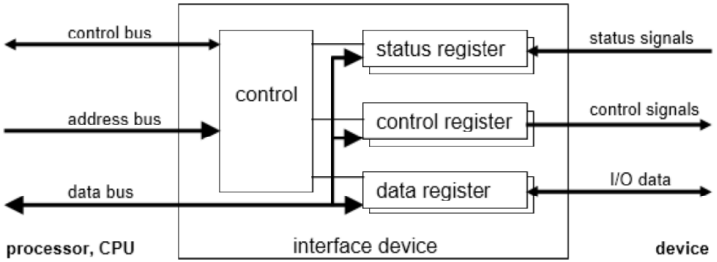
The storage of the hexadecimal number 0x0A0B0C0D = 168496141 results in a different byte order in memory, dependent of the microcontroller or peripheral hardware:



1. With **little-endian** hardware the least significant byte (**LSB**) is stored with the **lowest memory** (byte) **address**,
2. With **big-endian** hardware the most significant byte (**MSB**) is stored with the **lowest memory** (byte) **address**

I/O Synchronization

The communication between peripheral devices and the processor is performed by interface devices. These **interface devices** also define the base for synchronization between the processor, and devices operating at different speeds using certain registers.

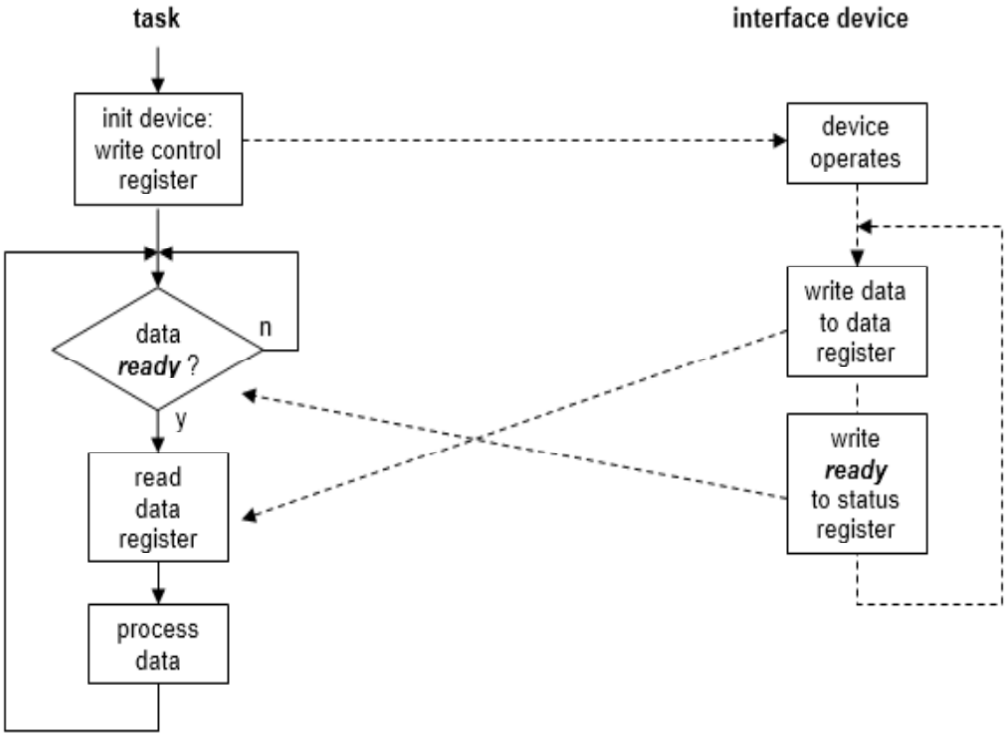


The typical components of an interface device are:

1. **Data register** to read/write the data to be received/transmitted.
2. **Control register** for configuring the device (e.g. operating mode, speed of serial ports, UART).
3. **Status register** reflecting the devices current state (e.g. sent/received data bits, ADC).

Polling

With the simplest synchronization method polling a task reads the status register continuously in a loop:



After initialization (by setting the control register) the task is waiting in an active loop on a status flag which is a particular bit within the status register of the device indicating that data is available in the data register. The task reads the data from the data register.

Advantages

- Simple synchronization type

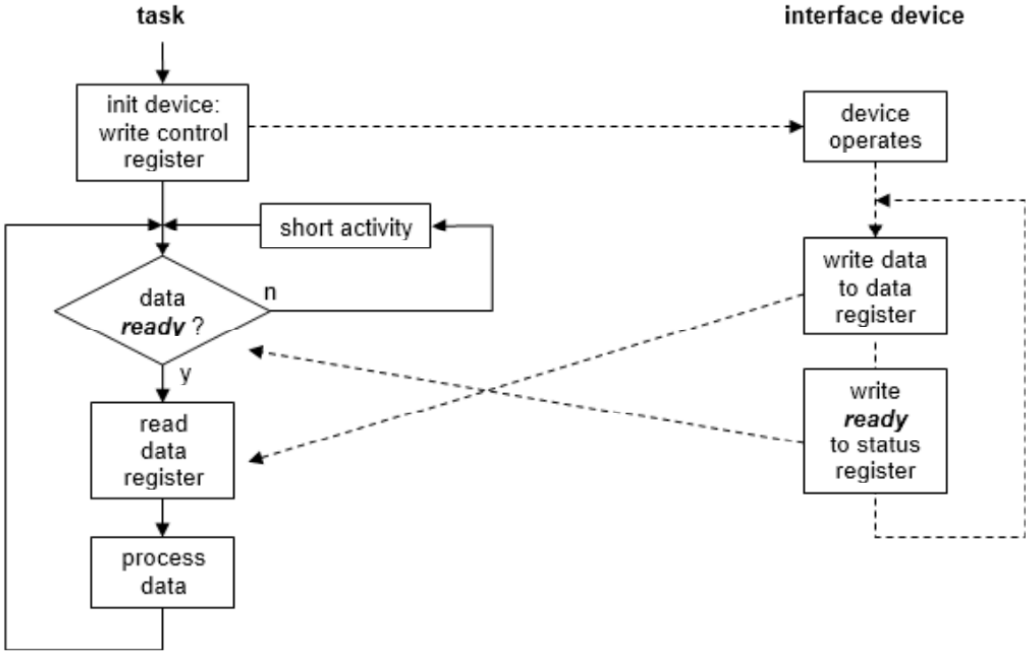
Disadvantages

- Consumption of processor time for the active loop
- Slow response times when multiple devices within a task are polled simultaneously.

Conclusion: Overall, polling is usually not favourable, due to inefficient CPU usage.

Busy Polling

With busy polling, the properties of simple polling are improved with performing some activity (typically short) within the waiting loop:

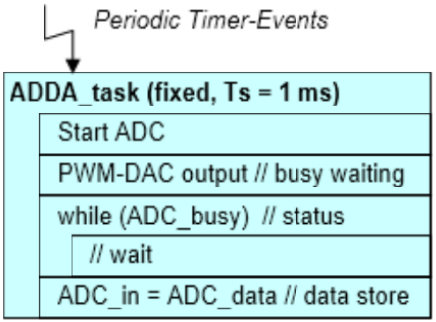


A disadvantage is, however, the response time slows down. Longer activities must be

split into smaller steps.

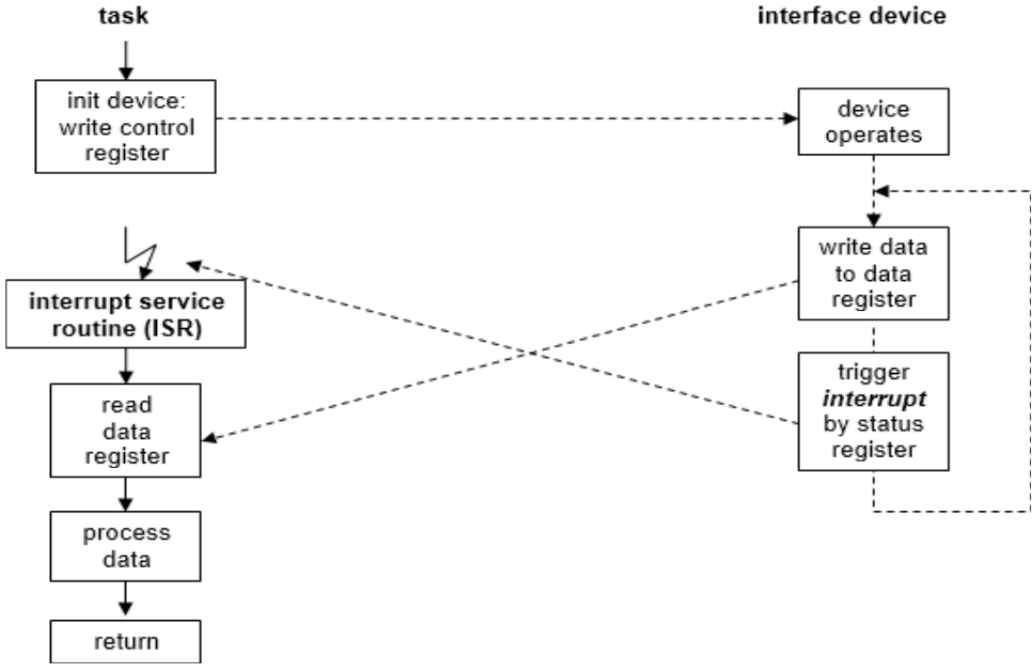
Example (sampling system (PWM-Control Experiment in Real-Time-Lab):

- Start an AD converter process with known duration,
- Offset / Gain correction of previously read values,
- Wait for the ADC to get ready,
- Readout of the ADC data register.



Interrupts

With synchronization by interrupts the interface executes an interrupt request with the processing CPU, which forces a currently running task to be interrupted in favor of a defined interrupt service routine **ISR**:



In the interrupt service routine (ISR), the data can be read from or written to the device.

Advantage:

- Processing time is consumed only, when data needs to be transferred

Disadvantage:

- Activating the ISR causes an overhead as the current processor state must be saved for the old task to resume after return of the ISR (equivalent to pre-emption with task-scheduling).
- To reduce the effect of overhead, larger amounts of data are transmitted in blocks, usually in conjunction with direct memory access (DMA).
- .The technique of synchronization with interrupts is very well suited for real-time applications and thus interrupts are widely used with micro-controllers with both synchronous and asynchronous real-time programming.

Interrupts and Task-Scheduling

The real-time operating system then has two options:

1. Interruption of the Task-Oriented Processing:

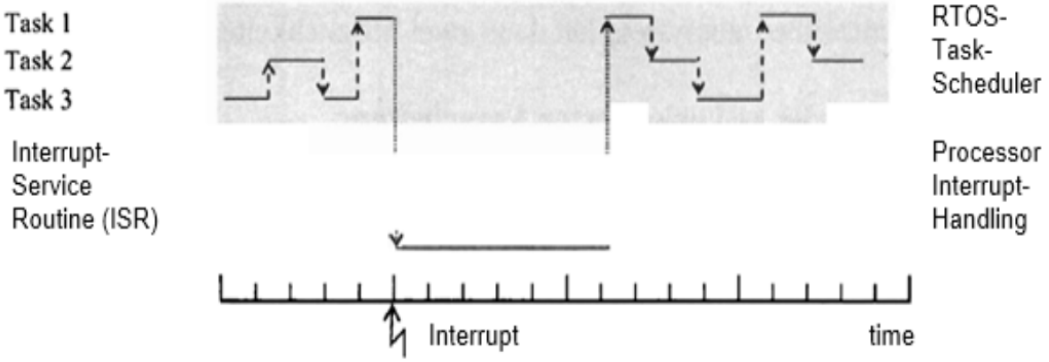
   - With an interrupt the currently running task is interrupted. The interrupt service routine (ISR) is executed immediately, after return the interrupted task continues.

   Advantage:

   - Very fast response to external events, because the interrupts are handled directly by the CPU hardware.

   Disadvantages:

   - The regular task-scheduling (eg EDF, FPP, TSS ...) is bypassed by direct handling of the processor hardware.
   - The prozessor uses its own priority based preemption (FPP), might conflicting with real-time scheduling



- Regardless of the scheduling strategy used by real-time scheduling events are treated with FPP scheduling, which can not guarantee 100% processor utilization (see section 2.2.7).
- This kind of direct processing of interrupts with interruption of the task-oriented processing is used for RTOS's with less performant microcontrollers, better solution: integration of interrupts with task-scheduling.- The total available processor utilization for task-scheduled processing is reduced from originally 100 % !

2. Integration of Interrupts with the Task-Oriented Processing:

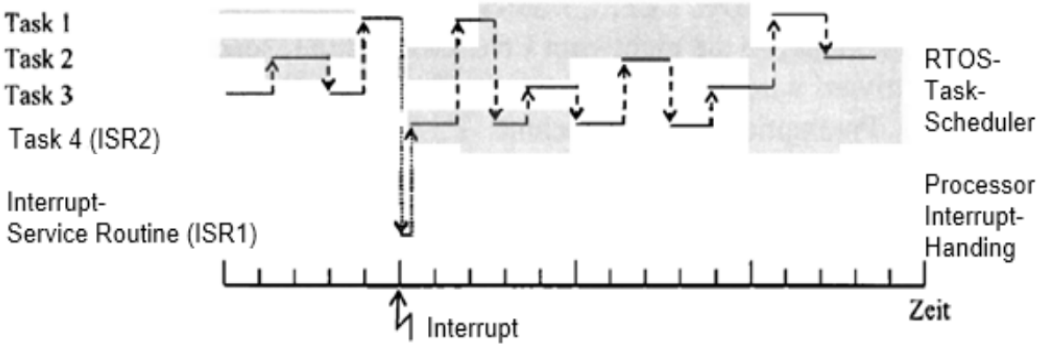
There is a separate task, integrated with task-scheduling, which is responsible for interrupt processing. Thus, whenever the processor gets a hardware interrupt, the ISR starts that separate task, responsible for interrupts.

Advantage:

- Interrupt handling is fully integrated with task-scheduling, there are no more two schedulers simultaneously, which might conflicting.
- An interrupt has the same scheduling policy as all other tasks. (e.g. an interrupt can have lower priority than the actually running task, such that it is not interrupted, with solution 1, this is not possible).

Disadvantage:

- The reaction to events is slowed, because the interrupt handling is not activated directly by hardware any more, but by the task-scheduler.



The integration of interrupts into the task-oriented processing is suitable for real-time operating systems using more powerful micro-controllers.

2.7 Specialized RTOS

Overview of some RTOS

Product	QNX	Posix.4	RT-Linux	VxWorks	OS-9	FreeRTOS	Windows CE	OSEK-OS
Producer	QNX Software	Posix	GPL-GNU Public License	Wind River	Microware	freertos.org Open Source	Microsoft	osek.org
Type	DS, RTOS	RTOS, ExtOS	ExtOS	RTOS	RTOS	MEBS	RTOS	MinOS
Target-System	IA32	IA32, IA64 PowerPC	IA32, PowerPC, ARM	IA32, PowerPC, 68k, div. µC	IA32, PowerPC, 68k	div. µC	IA32, PowerPC, MIPS ARM	div. µC
Languages	C, C++	C, C++, Java, Ada	C, C++, Java	C, C++, Java	C, C++, Java	C, C++	C, C++, Java	C
File-System	Unix, Windows	Unix	Unix, Windows	Unix, Windows	Windows	-	Windows	-
GUI	X-Win	X-Win	X-Win	X-Win	X-Win	-	Windows	-
Network	TCP/IP	TCP/IP, UDP/IP	TCP/IP, UDP/IP	TCP/IP	TCP/IP	TCP/IP	TCP/IP	-
Fieldbus	-	-	CAN	CAN, Profibus ?	CAN, Profibus, Interbus S	CAN	-	CAN, LIN, FlexRay
Scheduling	FPP,FPN, Timeslice, FIFO	FPP,FPN, Timeslice, User-def.	FPP,FPN, Timeslice	FPP,FPN, Timeslice	FPP,FPN, Timeslice	FPP,FPN, Timeslice	FPP,FPN	FPP,FPN, Timeslice

Classification of Real-Time Operating Systems

For classifying RTOSes there can be made a separation into five basic types:

1. **Minimal Real-Time Operating System (MinOS)**
   - Simple RTOS for microcontroller with limited memory resources.
   - RTOS is a library, to be linked with the application.
   - Simple I/O-functions No memory management, physical addressing only (no protected mode).
   - threads only (lightweight processes)
2. **Controller System (CS)**
   - Minimal real-time operating system
     - File system
     - Better error handling.

3. **Dedicated System (DS)**
    - Controller System
        - Memory management.
        - Protected mode
4. **Standard OS Extension (ExtOS)**
    - Standard OS (Windows, Linux, ..)
        - Some RTOS features
5. **Universal Real-Time OS (RTOS)**
    - Fully featured RTOS.
        - Features of standard OS.

Selection Criteria

A first selection criterion is the classification described in 2.6, which is set by the scope and functionality of a real-time operating system (RTOS).Other criteria selecting an RTOS are:

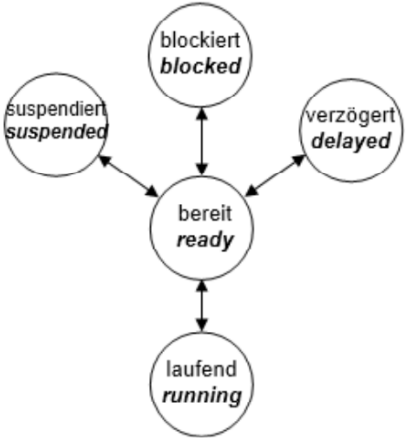
1. **Development and Target Environment**
2. **Modularity and Core Size**
3. **Adaptability**
4. **General Features**
5. **Performance**

VxWorks

VxWorks, by Wind River Company [2004], is a general purpose real-time operating system. In in its basic form, it supports no virtual memory (→ advanced product VxVMI). VxWorks uses heavyweight processes as tasks. However, task-switch was optimized to enable fast task switching. The context VxWorks stores in a task-control block (TCB) includes:

1. The program counter of the task,
2. The contents of processor registers and floating-point,
3. The stack of the dynamic variables,
4. The associated I / O devices,
5. The signal handler and
6. Various debugging information

The VxWorks state model for the tasks differs not much from the general model from section 2.1.3:



1. **Running** the task is executed (run).
2. **Ready** the task is ready to execute.
3. **Blocked** (pending) the task is waiting for the release of a resource and is blocked.
4. **Delay** (delayed) the task is stopped for a certain period of time.
5. **Suspended** (suspend) the task was suspended. This is intended for debugging purposes only.

By default, VxWorks uses FPP scheduling. Each task is assigned an initial priority. This

priority can be adjusted during runtime (function **taskPrioritySet**()). VxWorks defines 256 priority levels, with

1. level **0** corresponding to the **highest**, and
2. level **255** to the **lowest priority**.

The VxWorks task scheduler can be turned on and off explicitly by two functions (**taskLock() and taskUnlock()**). The switch will prevent, that the current task is pre-empted by a higher priority task realization of non-preemptive scheduling (FPN).

However, a task can still be interrupted by hardware interrupts. Deactivation of the task-scheduler is called **pre-emption locking**. It can be used for the protection of (short) critical sections. VxWorks also supports Time Slice Scheduling for tasks with equal priority.

Priority inversion

With priority based scheduling the priority inversion problem can arise, which became popular with the mars pathfinder mission ”Sojourner”, 1997):

Solution: Priority Inheritance [1].

The priority-inheritance protocol assures that a **task that holds a resource executes** at the **priority of the highest-priority task blocked** on that **resource**. In VxWorks the priority-inheritance protocol can be activated for a mutual-exclusion semaphore with the option **SEM_INVERSION_SAFE** enabled.

For communication between tasks VxWorks provides the following mechanisms:

1. Shared memory
2. Signals
3. Semaphore
4. Message pipes
5. Sockets to communicate with other computers via network
6. Non-local procedure calls (Remote Procedure Calls) to other computers over the network.

With VxWorks it is also possible to use libraries or program routines for several tasks simultaneously (shared code) **reentrancy techniques**

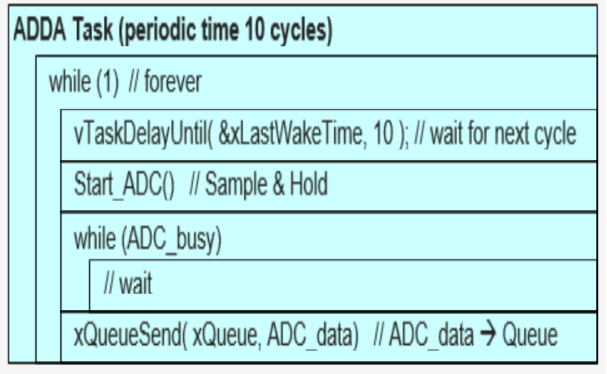
FreeRTOS

FreeRTOS is an Open-Source real-time operating system for Embedded Systems. FreeRTOS was ported to a large number of different micro-controllers with different performance.

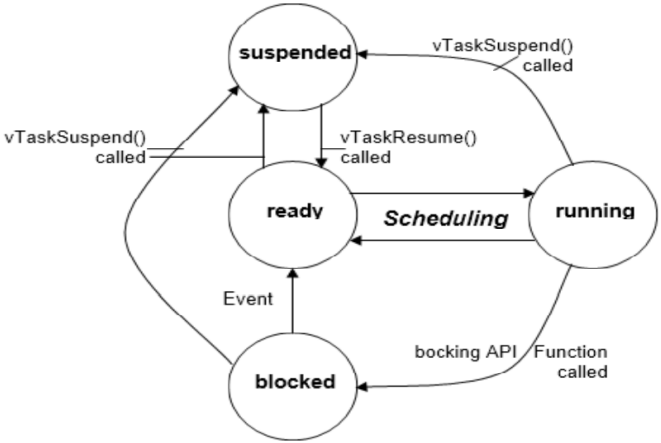
- Microcontrollers with ARM7-Architektur
- Microcontrollers of the ARM Cortex-M family (M0 . . . M4)
- Altera Nios II Softcore Processor
- Atmel AVR and Atmel AVR32
- Freescale Semiconductor HCS12-Familie and Coldfire V2
- Xilinx MicroBlaze and PowerPC PPC405
- Texas Instruments MSP430

Features

To ensure good maintainability, FreeRTOS is developed mostly in C, only a few functions are implemented in assembler. The scheduler can be configured for preemptive and co-operative operation. The OS (since version 4) supports two different task classes: ”real” tasks and coroutines, the latter are recommended where little space is available.

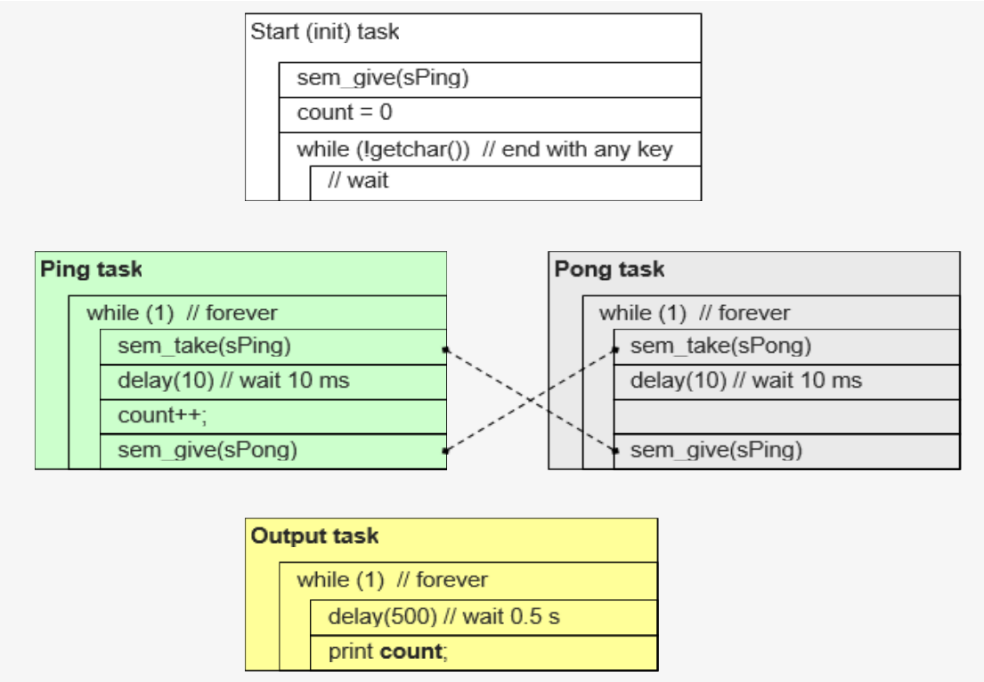


FreeRTOS Task-Model



Lab Experiment PingPong

Task diagram for 4 parallel running tasks (3 + 1 init):



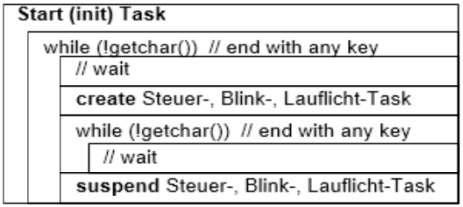
```
1  // ------------------------------------------------------------------------
2  /// \file    PingPong.c
3  /// \brief     RTP-Lab intro.
4  /// \author    Wolfgang Schulter
5  /// \license   for educational purposes only, no warranty, see license.txt
6  /// \date    14.01.2013 ws:  initial version
7  // ------------------------------------------------------------------------
8
9  #include "PingPong.h"          // PingPong module header
10
11 xSemaphoreHandle semPing;      // semaphore Ping
12 xSemaphoreHandle semPong;      // semaphore Pong
13
14 uint16_t count = 0;          // count variable incremented by ping
15
16 #define TICK_RATE_KHZ (configTICK_RATE_HZ/1000)
17 uint16_t delay1 = 10*TICK_RATE_KHZ;   // number of ticks <=> 10 ms delay as default
18 uint16_t delay2 = 500*TICK_RATE_KHZ;  // number of ticks <=> 500 ms delay as default
19 uint8_t do_printf = 0;         // print task: printf cout variable to stdout, if 1
20                       // = classic mode off by default (we have the GLCD)
21
22 // ------------------------------------------------------------------------
23 void init_PingPong()
24 {
25   vSemaphoreCreateBinary(semPing);      // <---- init to 1
26   vSemaphoreCreateBinary(semPong);      // <---- init to 1
```

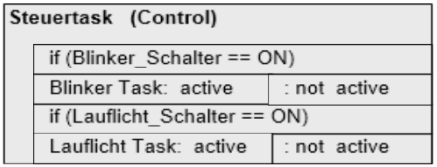
```
27    xSemaphoreTake(semPong, portMAX_DELAY);   // set to 0
28  }
29
30  // ---------------------------------------------------------------------------
31  void vPingTask(void * pvParameters)
32  {
33    wcet_init(&BWCET_PING);   // <---- wcet init
34
35    while(1) {
36
37      xSemaphoreTake(semPing,portMAX_DELAY);  // <---- block forever until ...
38      wcet_t1(&BWCET_PING); // <---- wcet measure
39
40      vTaskDelay(delay1);
41      xSemaphoreGive(semPong);  // <---- unblock pong task
42      count ++;
43
44      wcet_t2(&BWCET_PING); // <---- wcet measure
45    }
46  }
47
48  // ---------------------------------------------------------------------------
49  void vPongTask(void * pvParameters)
50  {
51    wcet_init(&BWCET_PONG);   // <---- wcet init
52
53    while(1) {
```

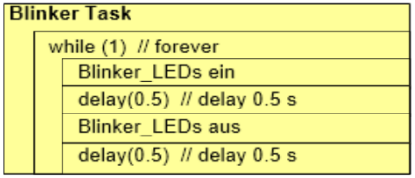
```
54
55      xSemaphoreTake(semPong,portMAX_DELAY);  // <---- block forever until ...
56      wcet_t1(&BWCET_PONG); // <---- wcet measure
57
58      vTaskDelay(delay1);
59      xSemaphoreGive(semPing);  // <---- unblock ping task
60
61      wcet_t2(&BWCET_PONG); // <---- wcet measure
62    }
63  }
64
65  // ---------------------------------------------------------------------------
66  void _print()
67  {
68    _dbg[0] = count;      // glcd debug variable _dbg[0]
69
70    if (do_printf) {      // since 0.5: classic mode is off by default
71      sprintf(_db, "%d ", count); DB;   // print to stdout
72    }
73  }
74
75  // ---------------------------------------------------------------------------
76  void vPrintTask(void * pvParameters)
77  {
78    wcet_init(&BWCET_PRINT);  // <---- wcet init
79
80    while(1) {
```

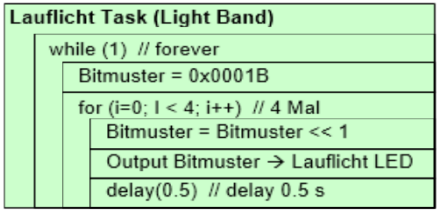
```
81      vTaskDelay(delay2);   // wait
82      wcet_t1(&BWCET_PRINT);  // <---- wcet measure
83
84      _print();       // GLCD update, print
85      count = 0;        // reset count variable
86
87      LED_TOGGLE(1);      // Toggle LED1
88
89      wcet_t2(&BWCET_PRINT);  // <---- wcet measure
90    }
91  }
```

Lab Experiment Multitasking









Lab-Experiment ***Multitasking***: The following tasks run (quasi) simultaneously:

1. Start:
   TaskGeneration of **Control**-, **Blinker**- and **Lauflicht**-Task. Suspend all tasks after receiving a char from the keyboard.
2. Steuer_Task:
   Suspend and resume **Blinker**- and **Lauflicht**-Task depending on the switch position of the switches **"Blinker_Schalter"** and **"Lauflicht_Schalter"**.
3. Blinker_Task:
   Switch on/off the LEDs in LED_Feld_1 with a periodic time 500 ms.
4. Lauflicht_Task:
   Shift-through an active LED from right to left each 500 ms.

OSEK (AUTOSAR OS)

OSEK means ”Open systems and interfaces for automotive electronics”, it stands for an industrial standard. OSEK is a trademark of Continental AG (up to 2007 from Siemens).

The 1993 committee consists of developers several car manufacturers, their suppliers and software services. Founding members were BMW, Daimler-Benz, Opel, Volkswagen, Bosch, Siemens and the Institute for Industrial Information Technology of the University of Karlsruhe (TH).

The work of the OSEK committee is continued since 2003 under the **AUTOSAR** project www.autosar.org.

The task-scheduler of the OSEK OS distinguishes between

- **Basic Tasks** run continuously until they are completed, or, until the OS switches to another task, or, when an interrupt occurs.
- **Extended Tasks** can have a waiting state, by a call of **WaitEvent()** allowing the processor to be released for other tasks

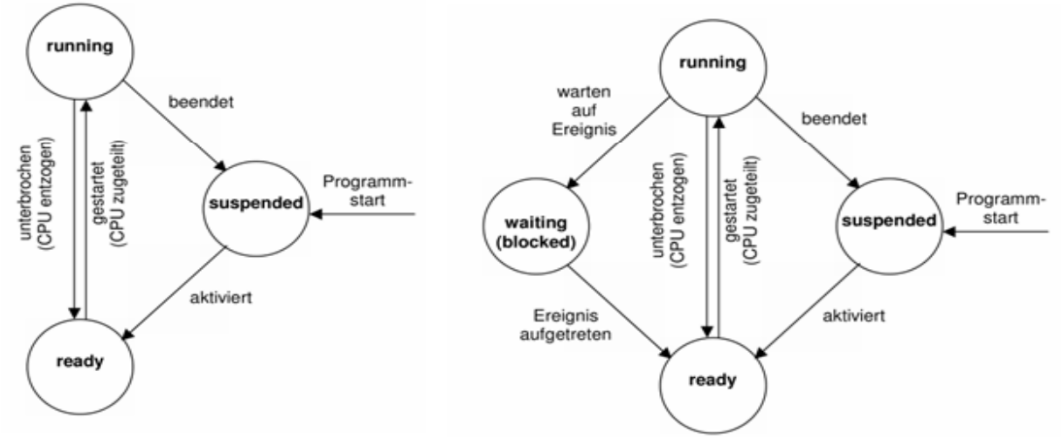
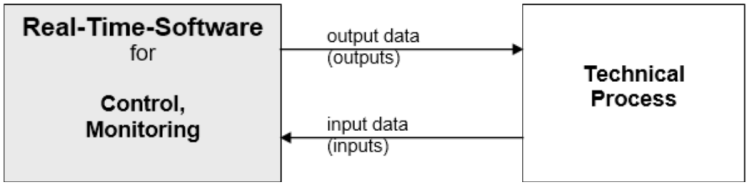


Figure 2.7.1: Task states of Basic Tasks (left) and an Extended Task (right)

Extended tasks can change into the state of **waiting** (**blocked**), if they have the processor, but can not continue because they have to wait for an event (e.g. the controller sends a CAN message and wait for the confirmation).

3 Real-Time Programming

Real-time programming is about the acquisition and the processing of process variables of a technical process by means of digital programmable processors.

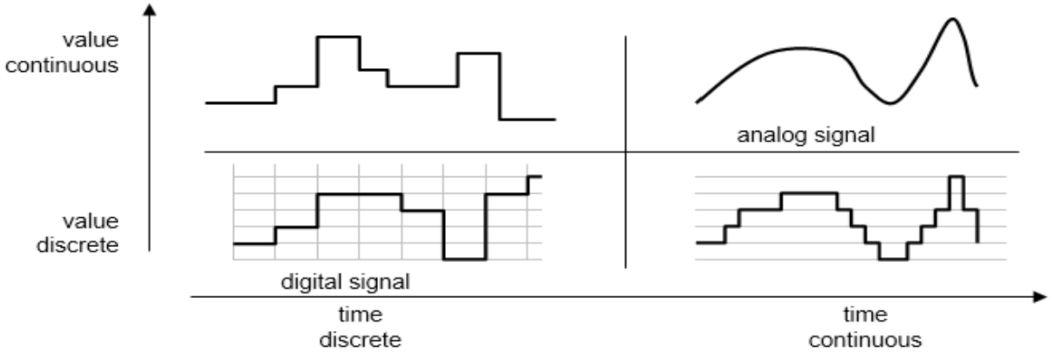


For the acquisition of physical quantities, there are sensors, which convert a physical quantity to be measured into an electrical signal (a function of time).

Analog signals, are **continuous in value** (real valued) and they can be **continuous in time**, or **time discrete**.

Digital signals are **discrete in value**, and either **time discrete** or **continuous in time**.

Since **computers** can **process time discrete** and **value discrete signals** only, analog **sensor input signals** have to be converted into digital (**time and value discrete) signals**.

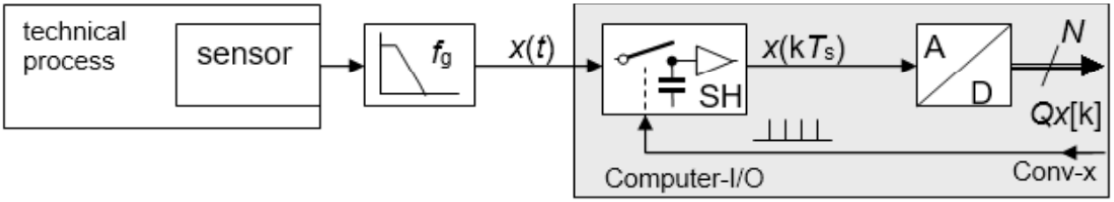


Since a process can be affected by actuator, the **digital**, **time discrete** control signal, produced by a real-time computer program, must be converted back into an analog signal.

3.1 Signal Conversion

An analog sensor signal $x(t)$ (time and value continuous) is first made time discrete by a sample-and-hold element. The sampling times are given from the I/O system of the computer. Analog signals are sampled at a constant sampling frequency f_s with sampling period $T_s = 1 / f_s$.

It is usually ensured that the signal $x(t)$ has no higher frequency components some limit frequency $f_g \leq 0.5\hat{A}\cdot f_s$. This is indicated by the analog low-pass filter with cutoff frequency f_g sampling theorem see section 3.1.1.



The amplitudes of the signal $x(k\hat{A}\cdot T_s)$ are still value continuous (real valued), but values change only at discrete times $k\hat{A}\cdot T_s$ (= **sampling**).

In a second step, the time discrete signal $x(k\hat{A}\cdot T_s)$ is converted into a value discrete signal using an ADC (see section 4.5), i.e. the real value x is approximated by an integer number Qx, out of a sequence (0 ... N-1), such that

$x \approx Q_x\hat{A}\cdot c_x$ with c_x as some conversion factor → (3.1).

After that $Qx[k]$ is a digital signal (time- and value discrete) which can be stored and processes in the computer. With increasing time, the index k increases, e.g. the sequence { Qx[k-2], Qx[k-1], Qx[k] } is the sequence of the last two recent and the actual sampling value. Example:

{	Qx[k-2],	9	5	-1
	Qx[k-1],	5	-1	2
	Qx[k]	-1	2	7
}		k=0	k=1	k=2 ...

a sequence $Qx[k]$ as actual sampling value and with $Qx[k\text{-}1]$ as the previous and $Qx[k\text{-}$

2] as the pre-previous sample.

In the computer, the value discrete sequence Qx[k] can be identified with the value continuous, time discrete sequence x[k] and thus the analog signal $x(t)$.

The quantization error $e = x - Q_x$Â·c_x with the sampling process can be minimized, since high resolution AD- and DA-converters are available $\rightarrow$ e can be usually neglected.

Digital Sampling Systems

Digital sampling systems are suitable for processing analog signals $x(t)$ with discrete values as a time discrete sequence $x[k] = x(k \cdot T_s)$.

The original analog signal $x(t)$ can be uniquely reconstructed from the time discrete sequence $x[k]$, i.e. the original analog signal has a unique representation by the discrete sequence $x[k]$:

Sampling Theorem of Shannon / Kotelnikov

> An analog signal sampled at constant frequency fs can be reconstructed from the discrete-time sequence of samples (up to a finite delay) if the bandwidth B is not greater than half the sampling frequency, thus $B < 0.5$ fs.

This is the fundamental theorem of digital signal processing.

If the bandwidth condition is violated, i.e. the bandwidth is greater than half the sampling

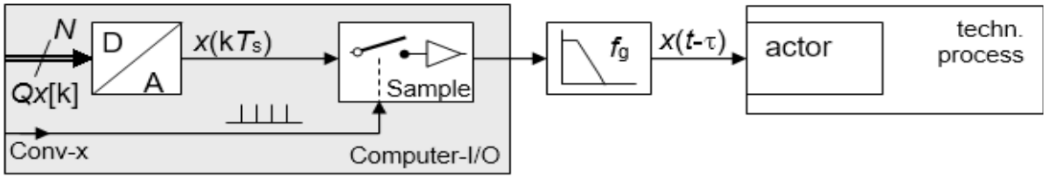
frequency, the analog signal can not be uniquely reconstructed from the sampling, due to irreparable signal distortion, known as **aliasing**.

A digital sampling system, sampling at frequency f_s must ensure that an analog signal is bandlimited before sampling an upper frequency limit $f_g \leq 0.5\ f_s$ .0.5 f_s is also called **Nyquist frequency** [3]. Thus, the sequence clearly determines the associated analog signal, which couldn't be processed by a digital computer directly!

(The proof is with another theorem of Fourier transform, stating that periodic sampling of a continuous time signal results in a periodic spectrum).

From the sampling theorem follows how the analog signal x(t-) can be reconstructed from the discrete time sequence *x*[k] by lowpass filtering of the discrete time signal:

The reconstruction is done by a periodic pulse train, sampling the DA converted, thus value discrete signal, which is finally (analog) lowpass filtered at cutoff frequency f_g =



0.5.f_s

The output of the so-called ”reconstruction lowpass” is $x(t\text{-})$, which is apart from a certain constant time delay the analog signal $x(t)$!

Real-Time Conditions for Sampling Systems

It is of great importance importance for signal quality, that strict periodicity of the sampling period. Any deviations from strict periodicity acts as signal interference, which may have some unpredictable consequences with closed loop sampling controllers (e.g. instable control).

A typical requirement for max. **jitter** (average deviation) of sampling period is 0.1% to 1%.

With digital sampling systems

1. strict periodicity of the sampling process in both the AD and the DA conversion
2. to comply with the Nyquist condition (the analog lowpass filters - before any ADC and -after any DAC- with the right cutoff frequency)

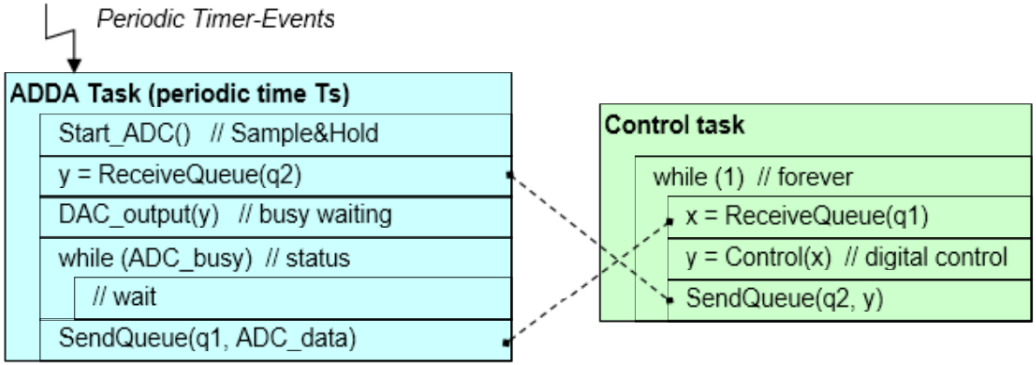
shall be respected.

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

Action	**Time Condition**
input of a sample of an analog signal	
computation of a signal sample	
output of the calculated signal sample	

Example: Sampling Control (Digital Control) as real-time application using 2 Tasks and

2 Queues



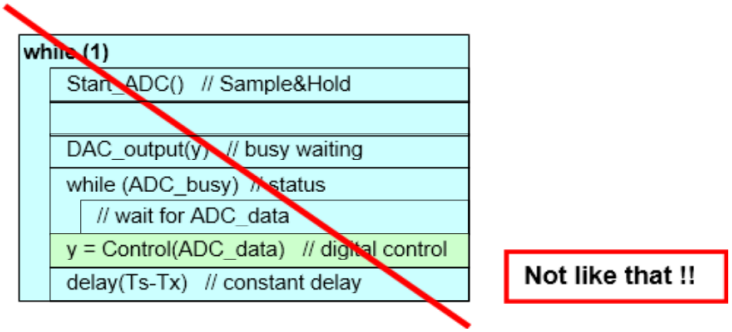
*the Queue q2 must be initialized, before the ADDA Task starts.

Instead of a queue, often just a semaphore is used for synchronization

Negative Example: Not appropriated due to differences in run-time jitter !!

- → Non deterministic sampling
  Violation of the exact real-time condition !!

- → Samples cannot be reconstructed,
  Distortions and Artefacts with (HQ) Multimedia signals
- → Stability of a closed-loop control is in danger, cannot been guaranteed !!



Quantization

A discrete-time signal $x(k\hat{A}\cdot T_s)$ is quantized using an AD converter, i.e. its value (amplitude) is approximated by an integer number with N discrete values. This is because a digital computer can handle value discrete (integer) values only, not value-continuous (real) values.

The number of values N are often powers of two, i.e. $N = 2^M$, so one obtains an M-bit AD conversion, with the conversion factor c_x, which gives the value of an LSB (least significant bit) as a physical quantity

$$\begin{aligned} x_Q &= Qx \cdot c_x + x_{\min} = Qx \cdot \tfrac{x_{\max}-x_{\min}}{N} + x_{\min} \\ c_x &= \tfrac{x_{\max}-x_{\min}}{N} = \tfrac{x_{\max}-x_{\min}}{2^M} \end{aligned} \tag{3.1}$$

Qx integer representation of x
xQ quantized physical quantity

cx conversion factor

In automation 8-, 10-, 12-, 14- bit AD converters are used more often than, say, 16-, 20- or 24-bit ADCs, the latter are mainly used for audio signals.

The deviation e = x_Q - x is called quantization error. In most applications, the number of conversion stages is N are chosen high enough, such that the quantization error is negligible, then $x_Q = x$.

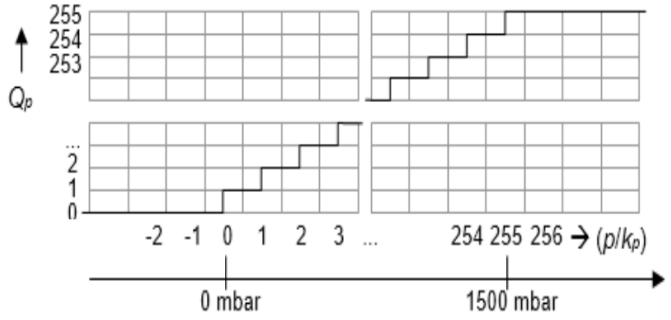
Example: A pressure sensor with an 8-bit AD converter is equipped for a measuring a pressure signal with a range of values (p_{min}, p_{max}) = (0, 1500) mbar.

The conversion factor by (**??**) is c_p = 1500 mbar / 2^8 = 5.86 mbar.

The quantization of the sensor signal is a step function with a linear increase. The value change takes place usually with a so-called ***mid tread*** pattern, i.e., in the middle of an

interval.

Q_p	p_Q/mbar
0	0
1	5.86
2	11.72
...	...
255	1494.1



The pressure signal is to be scanned by a process computer at a sampling frequency of 10 Hz. A warning signal at a low pressure p_L = 0.7 bar and at overpressure p_H = 1.2 bar shall be issued.

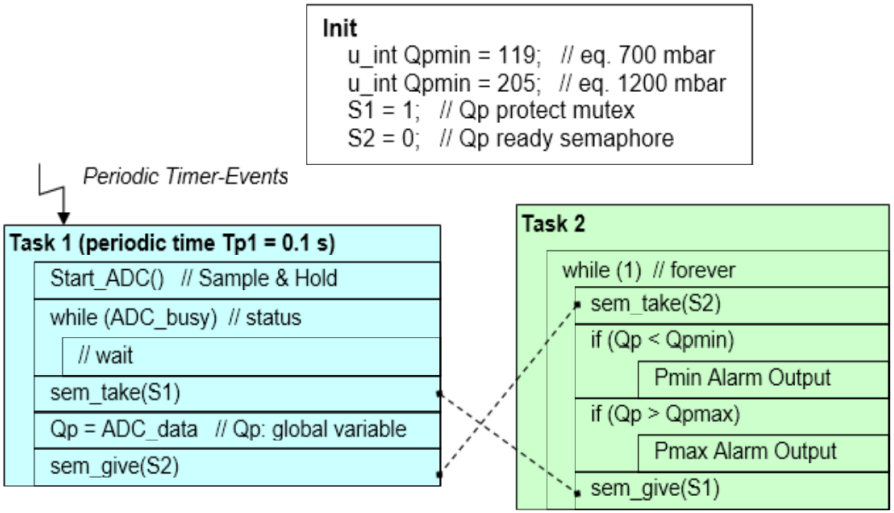
The bandwidth of the analog low-pass filter must be less than 5 Hz!

An appropriate real-time program for a microcontroller shall be designed with two tasks: 1. a task for pressure signal acquisition, 2. analysis and output.

Possible Solution: 2 Tasks in an FPN schedule:

- **task 1**: T_{p1} = 100 ms, (sampling jitter $<$ 0.1 ms). Acquisition of values *Qp*, setting the semaphore S1, if a new value Q*p*[n] is available.
- **task 2**: wait for S1, then
  (a) evaluate *Qp*[n]: compare with the lower limit
  Qp_{min} = round(p_{min} / k_P) = round(700 mbar / 5.86 mbar) = 119
  set Output_pmin, if (x[n] $<$ Qp_{min})

  (b) evaluate *Qp*[n]: compare with the upper limit
  Qp_{max} = round(p_{max} / k_P) = round(1200 mbar / 5.86 mbar) = 205
  set Output_pmax, if (x[n] $>$ Qp_{max})



3.2 Digital Filters

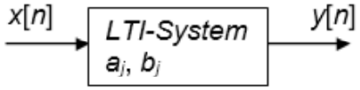
Linear digital filters can be described as LTI-Systems (**l**inear, **time-i**nvariant) by means of difference equations. The general form of a linear difference equation with constant, real coefficients a_j,b_j is

$$a_0y[n]+a_1y[n-1]+a_2y[n-2]+.....+a_Ny[n-N] = b_0x[n]+b_1x[n-1]+b_2x[n-2]+....+x_Mu[n-M]$$

$$\sum_{j=0}^{N} a_j \cdot y[n-j] = \sum_{j=0}^{M} b_j \cdot x[n-j]$$

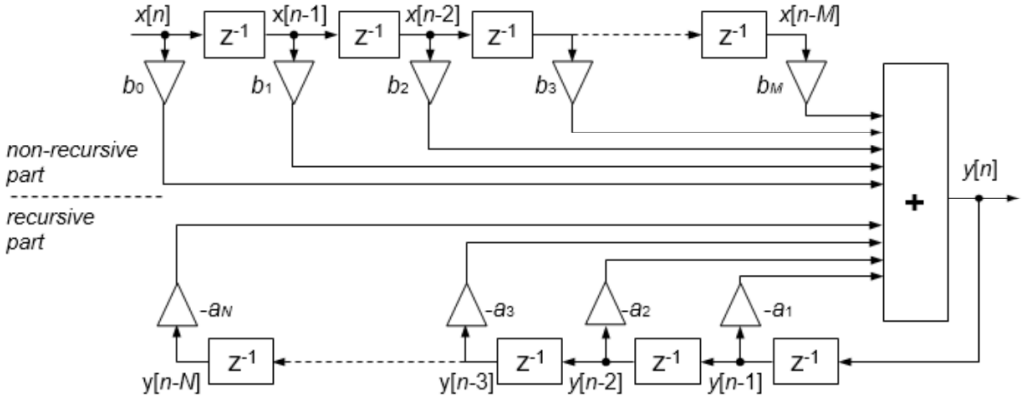
it is always possible to define $a_0 = 1$, which results in a recursion formula for the actual output value with constant coefficients a_j,b_j

$$y[n] = \sum_{j=0}^{M} b_j \cdot x[n-j] - \sum_{j=1}^{N} a_j \cdot y[n-j] \; with \; a_0 = 1 \; (byconvention) \qquad (3.2)$$



Structure of the general LTI-System

The z^{-1} rectangles symbolize a delay by one sample in time, the triangles symbolize a multiplication with a constant. The denominator degree N is also known as filter order.



There are 2 basic types of digital filters

1. recursive filters with **IIR**-behavior (**i**nfinite **i**mpulse **r**esponse), with $N > 0$
2. non-recursive filters with **FIR**-behavior (**f**inite **i**mpulse **r**esponse), with $N = 0$

With each new sampling period (increment of index k) a new output value $y[n]$ can be started earliest after the availability of a new input value $x[n]$, and the calculation needs to be finished before the next sampling period (Deadline !) see 3.1.1

For Integer implementations it is often economical to scale the difference equation (**??**) with am appropriate factor S such, that the multiply and add operations can be realized with integer arithmetic operations, most microcontroller CPUs support (rather than floating point arithmetic, which is common to larger, powerful CPUs):

$$y[n] = \frac{\sum_{j=0}^{M}(S\,b_j)\cdot x[n-j] \quad -\sum_{j=1}^{N}(S\,a_j)\cdot y[n-j]}{S} = \frac{A}{S}, \qquad S \in \mathbf{N} \qquad (3.3)$$

The coefficients Sb_j and Sa_j can be represented with sufficient precision by integer numbers, if S is chosen large enough. Since input- and output sequence values *x*[] and *y*[] are also integers, the numerator in (**??**) can be evaluated using integer-multiplication and -addition using an ***accumulator*** *A* with increased width (i.e. len x,y: 16-bits, len A: 64 bit).

Particularly effective as a choice for S = 2^{ldS} is a power of 2, thus the final division by N can be replaced by an equivalent right-shift operation

$$\frac{A}{S}(A >> ldS) \qquad without\ round\ op.$$

$$\frac{A + S/2}{S}(A + S/2) >> ldS \qquad with\ round\ op.$$

Rules for Optimizing Execution Time on Standard-Microcontrollers

Since most standard micro-controller CPUs do not support floating point arithmetic in hardware, and also lack for a full division (modulo) hardware support, these functions are implemented by means of software libraries.With critical CPU load, a code optimization is useful to reduce the WCET of any computationally intensive task. Particularly effective measures are

1. Avoiding add/sub and multiplication (float, double)
   →scaling by a power of two, and multiply with integers
   Exception: floating point operations are fine, if supported by the hardware.

2. Avoiding of Division (/)
   → Right Shift (> >) with powers of two instead of division

3. Avoiding of Modulo (%)

→ Bitwise AND (&) with powers of two instead of modulo

4. Review of the assembly based on the compiler list files (based on the specified CPU cycles) for execution time critical code

Example Digital IIR-Lowpass 1. Order

$$y[n] = b_0 \cdot x[n] \quad - a_1 \cdot y[n-1] \; y[n] = b_0 \cdot x[n] \quad - a_1 \cdot y[n-1]$$

Given: filter-coefficients: b_0 = 0.3333, a_1 = -0.6667.
Wanted: impulse-response and step-response for $n \geq 0$. C-realization with float- and integer- (fixed-point) arithmetic.

- **Impulse response** $x[n] = [n]$: $x =$ 1, 0, 0, 0, ...

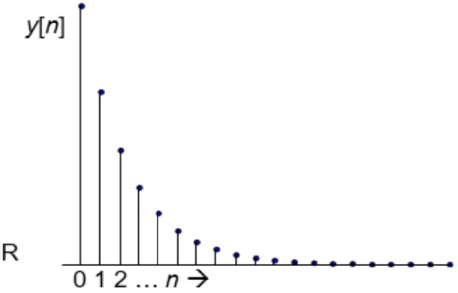
  Impulse-response is infinite long → IIR

- **Step-response** $x[n] = [n]$: x = 1, 1, 1, 1, ...

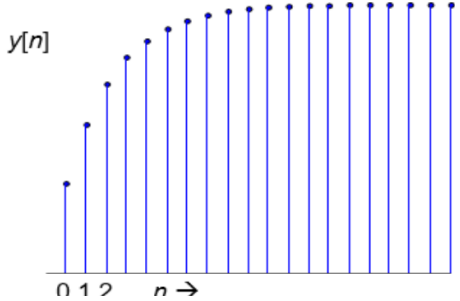
  → The step-response approximates the value 1.0 for large n, as expected.

n	x[n]	y[n]
0	1	0.3333
1	0	0.2222
2	0	0.1481
3	0	0.0988
4	0	0.0658
	…	

→ Impulse-response is infinite long → IIR

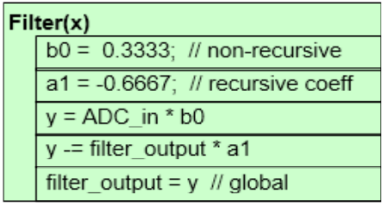


n	x[n]	y[n]
0	1	0.3333
1	1	0.5556
2	1	0.7037
3	1	0.8025
4	1	0.8683
	…	



C- Realization (float):

On microcontrollers without float-hardware needs hundreds of machine cycles and is thus very un-effective (section 3.2.1) !




```
// IIR Filter: y[n] = b0*u[n] - a1*y[n-1]
// ------------------------------------
short IIR1 float(short x)  // floating point realization
{
  const float b0 =  0.3333;
  const float a1 = -0.6667;
  float y = 0;
  extern short filter_output; // global variable

  y += x * b0;
  y -= filter_output * a1;

  filter output = y;
  return y;  // floating point: very unefficient !
}
```

C- Realization (integer):

Using the integer difference equation (**??**) with a power of 2: $S = 2^{ldS}$

$$S \cdot y[n] = (S\,b_0) \cdot x[n] - (S\,a_1) \cdot y[n-1]$$
$$S\,b_0 = S \cdot 0.3333$$
$$-S\,a_1 = S \cdot 0.6667$$

For example, if *ldS* = 12 is chosen, $S = 2^{12} = 4096$ follows

$$4096 \cdot y[n] = 1365 \cdot x[n] - (-2761) \cdot y[n-1]$$
$$S\,b_0 = 1365$$
$$S\,a_1 = -2731$$
$$y[n] = \frac{1365 \cdot x[n] + 2761 \cdot y[n-1]}{4096} = \frac{A}{4096} \approx (A >> 12)$$

Thus, with the scaled difference equation the *S*-fold result is calculated and stored in the accumulator *A*. Instead of dividing *A* by *S*, the effective right-shift operation (here: > > 12) is used.

Filter(x)
b0 = 0.3333; // non-recursive
a1 = -0.6667; // recursive coeff
y = ADC_in * b0
y -= filter_output * a1
filter_output = y // global

```
// ---------------------------------
// IIR Filter: y[n] = b0*u[n] - a1*y[n-1]
#define ldS 12
#define S   (1 << ldS)     // Skaling factor

short IIR1 int(short x)    // (fixed point) realization
{
  const int Sb0 =  0.3333*S;  // = 1365
  const int Sa1 = -0.6667*S;  // = -2731
  int A;                      // akkumulator
  extern short filter_output; // global variable

  A = x * Sb0;                // int promotion
  A -= filter_output * a1;    // int promotion

  filter output = (A >> ldS); // avoid div !
  return filter_output;
}
```

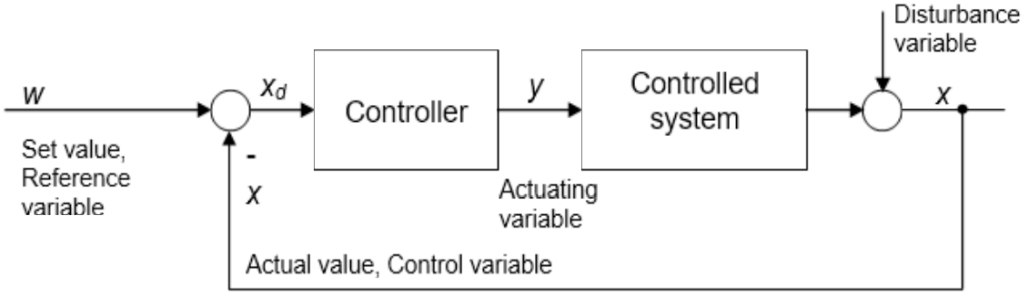
Even on microcontrollers without float-hardware, but hardware multiplyer for integers

utilizes only a handful of machine cycles and thus is very effective (on AVR ca. factor 100 faster than the float-version) !

3.3 Digital Controls

PID Controller

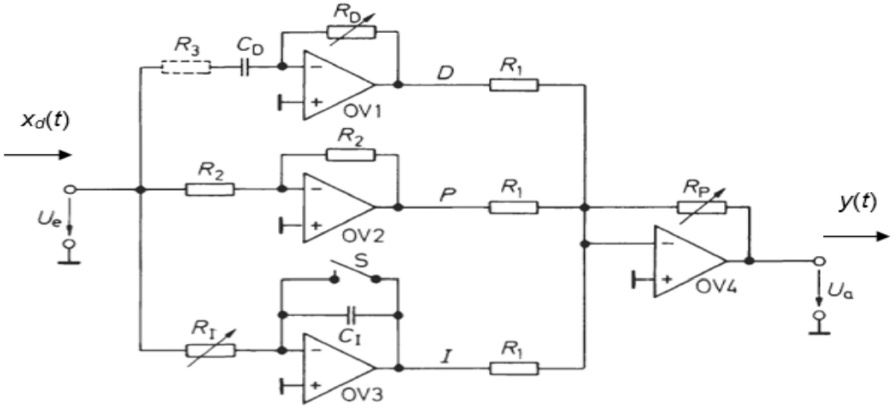
For many control systems in engineering, controls are used, which follow the so-called PID (**P**roportional-**I**ntegral-**D**ifferential) principle. Each of the 3 controller parts has its term with a specific parameter k_R, T_N, and T_V which allow the system response to be optimized with respect to stability, or under the influence of disturbances



System behavior of an analog PID Controller

$$y(t) = K_R \cdot \left[x_d(t) + \frac{1}{T_N} \int x_d(t)\, dt + T_V \frac{dx_d(t)}{dt} \right] \tag{3.4}$$

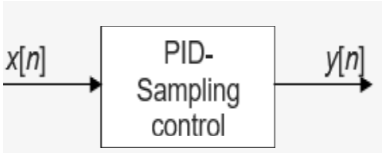
For an analog PID controller the circuit principle below was chosen in the past, the parameters were often calculated approximately. Finally, with the complete system, the parameters were finely adjusted empirically by adjustment of the balancing resistors.



dimensioning:$k_R = \frac{R_p}{R_1}$, $T_N = R_I \cdot C_I$, $T_V = R_D \cdot C_D$ [Tietze]

The PID controller behavior (**??**) can be simulated using a digital sampling system with

a suitable control algorithm invoked periodically with the sampling period T_s. The derivation is shown in [Oppen] oder [Kamm]:



Standard PID Regler Algorithmus ([LutzW] p. 477)

$$y[n] = y[n-1] + K_R \cdot \left[x[n] \cdot \left(1 + \frac{T_V}{T_S}\right) - x[n-1] \cdot \left(1 - \frac{T_S}{T_N} + 2\frac{T_V}{T_S}\right) + x[n-2] \cdot \frac{T_V}{T_S}\right] \tag{3.5}$$

PI-Controller

If the time constant T_V of the differential term is set to 0 in (**??**), the algorithm for the PI-controller is obtained, which has a proportional- and an integral term only:

$$y[n] = y[n-1] + K_R \cdot \left[x_d[n] - x_d[n-1] \cdot \left(1 - \frac{T_S}{T_N}\right)\right] \tag{3.6}$$

The PI controller type is quite popular, it can be realized with the PID-control algorithm.

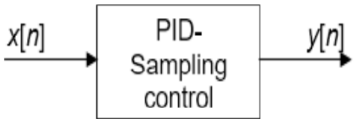
P-Controller

A P-controller has a proportional term only, with the simple algorithm

$$y[n] = K_R \cdot x_d[n] \tag{3.7}$$

PID-Controller as Digital Filter

The PID-control algorithm (3.5) can be expressed as a digital filter in



$$[y[n] = \sum_{i=1}^{M} a_i \cdot y[n-i] + \sum_{k=0}^{N} b_k \cdot x[n-k]] \tag{3.8}$$

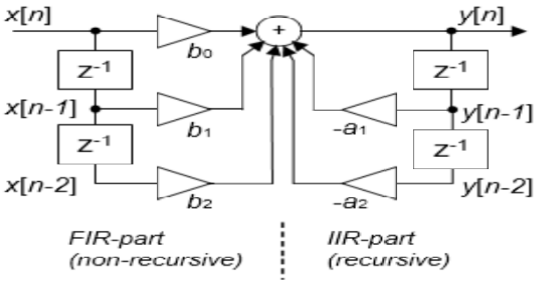
a common direct form (here, with input sequence $x_d[n] = x[n]$)

The coefficients for the des PID algorithm (**??**) thus define the coefficients of an IIR (Infinite Impulse Response) filter of 2. order

$$a_1 = -1,\ a_2 = 0,\ b_0 = k_R \cdot \left(1 + \frac{T_V}{T_S}\right),\ b_1 = -k_R \cdot \left(1 - \frac{T_s}{T_N} + 2\frac{T_V}{T_S}\right),\ b_2 = k_R \cdot \frac{T_V}{T_S}$$

$$y[n] = \sum_{k=0}^{2} b_k \cdot x[n-k] - \sum_{i=1}^{1} a_i \cdot y[n-i] + = b_0 \cdot x[n] + b_1 \cdot x[n-1] + b_2 \cdot x[n-2] - a_1 \cdot y[n-1]$$

A digital filter on 2. order in direct form can be represented as signal flow for time discrete signals (sequences), with triangles for multiplication with coefficients, the z^{-1} rectangles the delay elements for 1 sample and the circles represent additions.



The algorithm (3.8) calculates a new output value $y[n]$, for each new sampling period T_S with a new input value $x[n]$ and two previous input values $x[n\text{-}1]$ and $x[n\text{-}2]$ and the last output value $y[n\text{-}1]$, which is stored.

For calculating digital filters on microcontrollers, it is generally advantageous to use optimized libraries (if available), which respect the specifics of the processor CPU. With optimizing digital filter algorithms significant reductions in execution time can be achieved (see 3.2.1).

PID Algorithm in C

The algorithm (3.8) can be programmed in its general form in floating-point arithmetic easily with the C programming language:

- **IIR2**() realizes (3.8), which is the actual digital filter, and, buffering of the 2 past output values $y[n\text{-}1]$ and $y[n\text{-}2]$
- **store_xd**() realizes the control difference $x_d[n]$ from the current process value $x[n]$ and setpoint $w[n]$. store_xd() also realizes the buffering of the 3 controller input values $x_d[n]$, $x_d[n\text{-}1]$, $x_d[n\text{-}2]$
- **init_pid()** calculates from a given set of controller parameters k_R, T_N, und T_V with the constant sampling period T_S a set of filter coefficients a_pid[] und b_pid[] by (3.8)

```
// ----------------------------------------
float a_pid[3], b_pid[3], xd[3] = {0., 0., 0.}, y[3] = {0., 0., 0.}, w = 0.;
// ----------------------------------------
void init_pid(float a[], float b[], float kr, float Tn_Ts, float Tv_Ts)
{
  a[0] = a[2] = 0.0;  // Tn_Ts = Tn / Ts
  a[1] =  1.0;        // Tv_Ts = Tv / Ts
  b[0] =  kr*(1.0 + Tv_Ts);
  b[1] = -kr*(1.0 - 1.0/Tn_Ts + 2.0*Tv_Ts);
  b[2] =  kr*Tv_Ts;
```

```
}

// ----------------------------------------
void store_xd(float x0, float w)
{
  xd[2] = xd[1];    // xd*z^(-2)
  xd[1] = xd[0];    // xd*z^(-1)
  xd[0] = w-x0;   // xd*z^(0)
}

// ----------------------------------------
float IIR2(float x[],float y[],float a[],float b[])
{ // --- IIR 2nd order x[n] -> (a,b) -> y[n]
  float sum;

  // non-recursive (FIR) section
  sum = x[0]*b[0] + x[1]*b[1] + x[2]*b[2];

  // shift output buffer values
  y[2] = y[1];  // y*z^(-2)
  y[1] = y[0];  // y*z^(-1)

  // recursive (IIR) section
  sum += y[1]*a[1] + y[2]*a[2];

  return y[0] = sum;  // y*z^(0)
}
```

It should be noted, that the use of floating-point arithmetic in microcontrollers without hardware floating-point unit usually results in large execution times, and is therefore usually avoided entirely → use **integer arithmetic** (3.2.1) with less powerful microcontrollers !

As real-time software realization, it can be favorable, to calculate the FIR and the IIR filter parts by concurrent tasks, which can then be run simultaneously on a multiprocessor hardware.

Requirements for the Digital PID-Controller

Since this controller is a digital sampling system, it must respect the requirements of the sampling theorem on the strict periodicity with T_s with sampling of the control variable x and the output of the actuating variable y.

For the input signal, an AD converter, either working as SAR (usually integrated on microcontrollers) or as flash type, a good strategy for getting an ADC value within the strict timing must be found (starting conversion, waiting on the result).

1. Events with period T_s and with **minimal** deviation (**jitter**) must be produced definition of a timer event (Ts)
2. start of AD conversion for the **input** must occur immediately after the Ts-event, **minimal jitter**, (limited by the specified interrupt latency).
3. The **output** value must be DA converted (with a delay of exactly one sampling period), with minimal jitter aas with AD conversion.
4. It is important to ensure that the algorithm operates only with a **consistent set** of parameters and input and output buffers with consistent values.
5. The **controller parameters** must be **adjustable** from outside (e.g. by asynchronous bus event). The latency time allowed for a complete switch of the parameters is 100Â·T_s.It must be ensured, that any parameter change, always results with a stable digital filter.

6. The set value must be adjustable from outside (e.g. by asynchronous bus event). The latency time allowed here is 20Â·T_s.
7. The real-time program shall be executable with any type of task scheduling.

Task Definition

A real-time program with 4 tasks can meet the specified requirements:

- **Task T1**: a high-priority task, which controls the timing conditions for reading the input from the AD converter and writing the output to the DA converter. (wait on periodic timer event **Ev_Ts**)
- **Task T2**: a medium priority task, which performs the calculation of the output sample by the IIR algorithm, after a new input value is ready. Task 2 may run only when task 1 is finished → event **Ev_12**.
- **Task T3**: an asynchronous low-priority task, which performs the conversion and modified PID control parameters and provides a consistent set of IIR filter coefficients → event **Ev_par**.

- **Task T4**: an asynchronous low-priority task, which lets change the set value→ event **Ev_w**

Critical Data

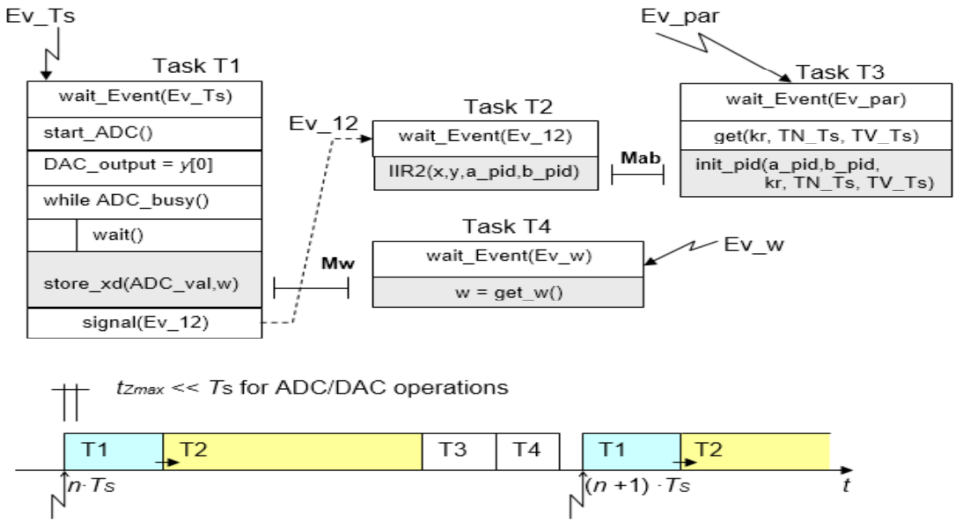
- **w (set value)**: read by T1 and can be concurrently written by T4 → Mutex **Mw**
- **a_pid, b_pid PID filter coefficients**: These are calculated and read by T2, concurrently written by T3 → Mutex **Mab**

Although the input buffer xd[], and the buffer of output values y[] is accessed by the two tasks T1 and T2, it is guaranteed by the order synchronization of T1-T2, that there will be no conflict, even without a mutex.

Realization of a Digital PID Control with 4 Tasks

This real-time program

- assures with T1: compliance with the strict time conditions for the two sampling systems ADC→$x[0]$ and $y[0]$→DAC: called periodically with minimal jitter.



- by **Ev_T12** is ensured, that T2 runs only when T1 is finished (cooperation sync)
- by Mutex **Mab is** ensured, that the PID algorithm is executed only with a consistent set of parameters.
- assumption is, that the utilization of task 2 was determined by measuring WCET to be $H_2 \approx 50\%$. If H_2 is too large, optimizations are necessary (e.g. integer arithmetic)

Exercise 1: Write C Code for Realizing Task T2 under FreeRTOS.

- develop by re-factoring PingPong, realize Ts = 2 ms

```
/* ------------------------------------------------------------------
   iir2.c: Module for realization of a digital filter, 2. order
   ------------------------------------------------------------------*/
// ----------------------------------------
float a_pid[3], b_pid[3], xd[3] = {0., 0., 0.}, y[3] = {0., 0., 0.}, w = 0.;
// ----------------------------------------
float IIR2(float x[],float y[],float a[],float b[])
{ // --- IIR 2nd order x[n] -> (a,b) -> y[n]
  float sum;
  // non-recursive (FIR) section
  sum = x[0]*b[0] + x[1]*b[1] + x[2]*b[2];
  // shift output buffer values
  y[2] = y[1];  // y*z^(-2)
  y[1] = y[0];  // y*z^(-1)
  // recursive (IIR) section
  sum += y[1]*a[1] + y[2]*a[2];
  return y[0] = sum;  // y*z^(0)
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
#include "FreeRTOS.h"
```

```
xSemaphore sEv_12, siir;                    // Semaphore sEv_12, Mutex siir
// ---------------------------------------------------------------------------
void vT2(void * pvParameters)           // re-factored vPingTask
{
  while(1) {
    xSemaphoreTake(sEv_12,portMAX_DELAY); // on Event Ev_12
    xSemaphoreTake(siir,portMAX_DELAY);   // acquire mutex
    IIR2(xd, y, a, b);              // Filter operation
    xSemaphoreGive(siir);           // release mutex
  }
}

// ---------------------------------------------------------------------------
void vT1(void * pvParameters)         // re-factored vPrintTask
{
  short ADC_in;
  portTickType xLastWakeTime;
  xLastWakeTime = xTaskGetTickCount();

  while(1) {
    vTaskDelayUntil(&xLastWakeTime, 2*configTICK_RATE_HZ/1000);   // exact periodic Ts
    = 2 ms
    ADC_start();            // start AD (--> SAR sample & hold) ...
    ...               // busy waiting stuff here (like DAC outputs)
    while(ADC_busy());      // wait for ADC (some microseconds) ...
    ADC_in = ADC_get();       // store ADC result
    update_x_buffer(ADC_in);  // update the filter input buffer xd[]
```

```
49      xSemaphoreGive(sEv_12);    // sync to task T2
50    }
51  }
```

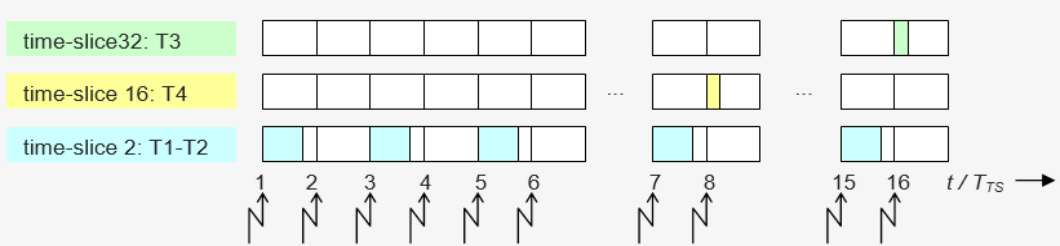
Exercise 2: PI controller (for ATmega microcontrollers)

1. Design a real-time program for a PI controller with the non-preemptive binary fixed time-slice method (FTS) for the following parameters
   execution times: T_{e1} = 50 μs, T_{e2} = 300 μs, T_{e3} = 100 μs, T_{e4} = 50 μs.
   - T3 und T4 shall be able to process asynchronous events about 20 times per sec.
   - the initial control parameters: k_R = 5, T_N = 20 ms.
2. Determine a new Schedule for the case where due to halving of the clock frequency of the micro-controller, each task has approximately twice the execution time of (1).

Solution:

1. H_1 = 5 %, H_2 = 30 %, H_3 = 0.1/20 = 0.5 %, H_4 = 0.05/20 = 0.25 % H = 30.75 %

   with 3 tasks, non-preemptive binary time-slice scheduler with $T_{TS} \geq \max(T_{ei})$,
   $\rightarrow$ periodic timer events with basic time-slice T_{TS} = 0.5 ms, so execution times of each of the 4 tasks are below the maximum possible execution time of the basic time-slice.




```
// ----------------------------------------
unsigned int t_Ts = 0;
void tSlice_schedule(void)  // every 0.5 ms
{
  // binary timeslice schedule
  if (t_Ts & 1) {
    // the 2 time slice: 2*TTs = Ts
    T1();  // Task T1
    T2();  // Task T2

  } else if ((t_Ts & 0x0F) == 8) {
    // the 16 time slice: 16*TTs = 8 Ts
    if (Ev_w) T4(); // async T4

  } else if ((t_Ts & 0x1F) == 16) {
    // the 32 time slice: 32*TTs = 16 Ts
    if (Ev_par) T3(); // async T3
  }
  t_Ts++;
}
```

t_Ts	16	8	4	2	1
	0	0	0	0	0
	0	0	0	0	1
	0	0	0	1	0
	0	0	0	1	1
	0	0	1	0	0
	0	0	1	0	1
	0	0	1	1	0
	0	0	1	1	1
	0	1	0	0	0
	0	1	0	0	1
	0	1	0	1	0
	0	1	0	1	1
	0	1	1	0	0
	0	1	1	0	1
	0	1	1	1	0
	0	1	1	1	1
	1	0	0	0	0
			0	0	1

- T1 and T2 run successively (no sync needed) in the time-slice called every 2 times
- T4 runs every 8 ms (= time-slice 16)

- T3 runs every 16 ms (= time-slice 32), all synchronous to the periodic timer events
- the execution times of all 4 tasks are well below the maximum time T_{TS} = 0.5 ms.

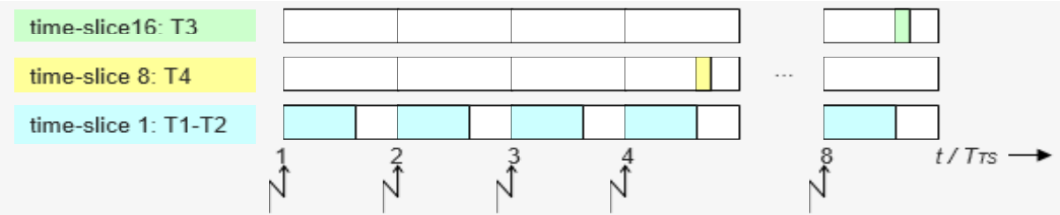
2. Doubled execution time (e.g. due to reduction of the CPU clock frequency)
   T_{e1} = 100 μs, T_{e2} = 600 μs, T_{e3} = 200 μs und T_{e4} = 100 μs

   H_1 = 10 %, H_2 = 60 %, H_3 = 0.2/20 = 1 %, H_4 = 0.1/20 = 0.5 % H = 71.5 %

   Solution:
   with 3 tasks, non-preemptive binary time-slice scheduler with $T_{TS} \geq \max(T_{ei})$,

   → periodic timer events with basic time-slice T_{TS} = 1.0 ms, so execution times of each of the 4 tasks are below the maximum possible execution time of the basic time-slice.

   - T4 runs every 8 ms (time-slice 8)




```
// ---------------------------------------
unsigned int t_Ts = 0;
void tSlice_schedule(void)  // every 1 ms
{
  T1();  // Task T1
  T2();  // Task T2

  // binary timeslice schedule
  if (t_Ts & 1) { // (t_Ts modulo 2) == 1
    // the 2 time slice: 2*TTs = 2*Ts

  } else if ((t_Ts & 0x07) == 4) { // (t_Ts modulo 8) == 4

    // the 8 time slice: 8*TTs = 8 Ts
    if (Ev_w) T4(); // async T4

  } else if ((t_Ts & 0x0F) == 8) { // (t_Ts modulo 16) == 8

    // the 16 time slice: 16*TTs = 16 Ts
    if (Ev_par) T3(); // async T3
  }
  t_Ts++;
}
```

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

- T3 runs every 16 ms (time-slice 16), both asynchronous to the periodic timer events (after T1-T2) !
- The maximum execution time is within time-slice 8: T_{e1} + T_{e2} + T_{e3} = 900 μs.

3.4 Finite State Machines (FSM)

In most embedded applications finite state machines are used to interface with non-synchronous, periodic devices, such as switches, slow actuators, indication lamps, networks . . .

Traditional event-based programming is used with modern GUI based standard OS, Windows, Linux, .. with both desktop and portable applications handling keyboard mouse, touchpanels, . . . as inputs as well as with networks.→ https://en.wikipedia.org/wiki/Automata-based_programming

Moore Finite State Machine: Each state has assigned exactly one output.

Coffee Automat: State Transient Table

Coffee Automat: Implementation in C (→ ProgC FSM_CoffeeAutomat)

```
// ------------------------------------------------------------------------
// the FSM states for the Coffee Automata
// ------------------------------------------------------------------------
enum {  FSM_STATE_IDLE,
        FSM_STATE_COFFEE,             FSM_STATE_COFFEE_MILK,
        FSM_STATE_COFFEE_SUGAR,       FSM_STATE_COFFEE_MILK_SUGAR,
        FSM_STATE_MAKE_COFFEE,        FSM_STATE_MAKE_COFFEE_MILK,
        FSM_STATE_MAKE_COFFEE_SUGAR,  FSM_STATE_MAKE_COFFEE_MILK_SUGAR,
};
// ------------------------------------------------------------------------
int fsm_state = FSM_STATE_IDLE;
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
// the FSM outputs, one for each state, (Moore Automata model)
// ------------------------------------------------------------------------
const int fsm_output[NUM_FSM_STATES] = {  // output is fsm_output[fsm_state]
    0,                          // bit0: add coffee
    0,  0,                        // bit1: add milk
    0,  0,                        // bit2: add sugar
    1,  3,
    5,  7
```

```
23  };
24
25  // ------------------------------------------------------------------------
26  // the FSM inputs
27  // ------------------------------------------------------------------------
28  enum {  FSM_INP_NONE=0, FSM_INP_LEFT, FSM_INP_RIGHT, FSM_INP_READY
29  // ------------------------------------------------------------------------
30  int fsm_tic(int fsm_inp)    // called periodically, e.g. each 50 ms
31  {
32    // -----------------------------------------------------------
33    // the FSM transients, realizing the state transient table
34    // -----------------------------------------------------------
35    switch (fsm_state) {
36    case FSM_STATE_IDLE:
37      if (fsm_inp == FSM_INP_LEFT) fsm_state = FSM_STATE_COFFEE;
38      if (fsm_inp == FSM_INP_RIGHT) fsm_state = FSM_STATE_COFFEE_MILK;
39      break;
40
41    case FSM_STATE_COFFEE:
42      if (fsm_inp == FSM_INP_LEFT) fsm_state = FSM_STATE_MAKE_COFFEE;
43      if (fsm_inp == FSM_INP_RIGHT) fsm_state = FSM_STATE_COFFEE_SUGAR;
44      break;
45
46    case FSM_STATE_COFFEE_MILK:
47      if (fsm_inp == FSM_INP_LEFT) fsm_state = FSM_STATE_MAKE_COFFEE_MILK;
48      if (fsm_inp == FSM_INP_RIGHT) fsm_state = FSM_STATE_COFFEE_MILK_SUGAR;
49      break;
```

```
50
51  case FSM_STATE_COFFEE_SUGAR:
52    if (fsm_inp == FSM_INP_LEFT) fsm_state = FSM_STATE_IDLE;
53    if (fsm_inp == FSM_INP_RIGHT) fsm_state = FSM_STATE_MAKE_COFFEE_SUGAR;
54    break;
55
56  case FSM_STATE_COFFEE_MILK_SUGAR:
57    if (fsm_inp == FSM_INP_LEFT) fsm_state = FSM_STATE_IDLE;
58    if (fsm_inp == FSM_INP_RIGHT) fsm_state = FSM_STATE_MAKE_COFFEE_MILK_SUGAR;
59    break;
60
61  case FSM_STATE_MAKE_COFFEE:
62  case FSM_STATE_MAKE_COFFEE_MILK:
63  case FSM_STATE_MAKE_COFFEE_SUGAR:
64  case FSM_STATE_MAKE_COFFEE_MILK_SUGAR:
65    if (fsm_inp == FSM_INP_READY) fsm_state = FSM_STATE_IDLE;
66    break;
67
68  default:
69    ;   // no state change
70  }
71  OUTPUT(fsm_output[fsm_state]);    // Moore FSM: one output per state
72  return
```

The above implementation in C is optimized for readability rather than for minimum run-

time or memory ressources. However, the above code uses very little memory space and is running quite fast.

Recipe for Realizing an FSM in C (Moore FSM)

- Analyze the problem before coding, plot a state diagram, define the states
- Use enums to encode states in a well readable form

- Define outputs as enums with suitable encoding, e.g. as bit-fields
- Define input conditions causing the state transitions as enums as well
- Realize a periodicaly called fsm_tic function, realize transients and outputs

Exercise: Develop a Traffic-Light including yellow phases from section 1.2 as a Moore-FSM as the above Coffee Automat with C-Code. The yellow phases shall last 2 s, the main green/red phases shall be 8 s each. Assume the fsm_tic() is called with 1 s periodic time. Plot a State Diagram and the state transient table.

C-Code (→ ProgC FSM_TrafficLight)

```
// ------------------------------------------------------------------------
// the FSM states, here for the TrafficLight
// ------------------------------------------------------------------------
enum fsm_state_e {
    FSM_STATE_1=0,            FSM_STATE_1Y,
    FSM_STATE_2,            FSM_STATE_2Y,
    NUM_FSM_STATES
};

// ------------------------------------------------------------------------
// the FSM state
// ------------------------------------------------------------------------
int fsm_state = FSM_STATE_1;  // initial state
int fsm_count = 0;          // the FSM counter

// ------------------------------------------------------------------------
// the FSM outputs, bit assignments, one for each state (Moore)
// ------------------------------------------------------------------------
enum fsm_out_e {  FSM_OUT_A1_G=1,   FSM_OUT_A1_Y=2,   FSM_OUT_A1_R=4,
             FSM_OUT_A2_G=0x10,  FSM_OUT_A2_Y=0x20,  FSM_OUT_A2_R=0x40,
             FSM_OUT_A3_G=0x100, FSM_OUT_A3_Y=0x200, FSM_OUT_A3_R=0x400,
             NUM_FSM_OUTPUTS };

const int fsm_output[NUM_FSM_STATES] = {        // output is fsm_output[fsm_state]
    FSM_OUT_A1_R|FSM_OUT_A2_G|FSM_OUT_A3_G,   // output with FSM_STATE_1
    FSM_OUT_A1_R|FSM_OUT_A1_Y|FSM_OUT_A2_Y|FSM_OUT_A3_Y,  // output with FSM_STATE_1Y
```

```
    FSM_OUT_A1_G|FSM_OUT_A2_R|FSM_OUT_A3_R,   // output with FSM_STATE_2
    FSM_OUT_A2_Y|FSM_OUT_A2_R|FSM_OUT_A3_R
          |FSM_OUT_A2_Y|FSM_OUT_A3_Y        // output with FSM_STATE_2Y
};

// ------------------------------------------------------------------------
#define T_RED_GREEN 8 // Time for Red/Green phases in Ts
#define T_YELLOW  2   // Time for Yellow    phases in Ts
// ------------------------------------------------------------------------
int fsm_tic()
{
  // ------------------------------------------------------------
  // the FSM
  // ------------------------------------------------------------
  if (fsm_count < T_RED_GREEN)
    fsm_state = FSM_STATE_1;
  else if (fsm_count < (T_RED_GREEN + T_YELLOW))
    fsm_state = FSM_STATE_1Y;
  else if (fsm_count < (2*T_RED_GREEN + T_YELLOW))
    fsm_state = FSM_STATE_2;
  else if (fsm_count < (2*T_RED_GREEN + 2*T_YELLOW))
    fsm_state = FSM_STATE_2Y;
  else {
    fsm_state = FSM_STATE_1;
    fsm_count = 0;  // modulo cycle time
  }
  // ------------------------------------------------------------
```

```
  fsm_out();          // Moore FSM: one output per state
  // ------------------------------------------------------------
  fsm_count ++;
  return fsm_state;
}

// ---------------------------------------------------------------------------
int main()
{
  int ch;
  gotoXY(0,0);
      printf("Traffic light, Ctrl-Break: exit");
      do {  // basic terminal loop
        ch = 0;
        if ( kbhit() ) {
            ch = getch();
        }
      fsm_tic();      // fsm tick
        Sleep(1000);    // Ts = 1000 ms

    } while ( ch != 127); // until user hits ctrl-backspace.
  return
```

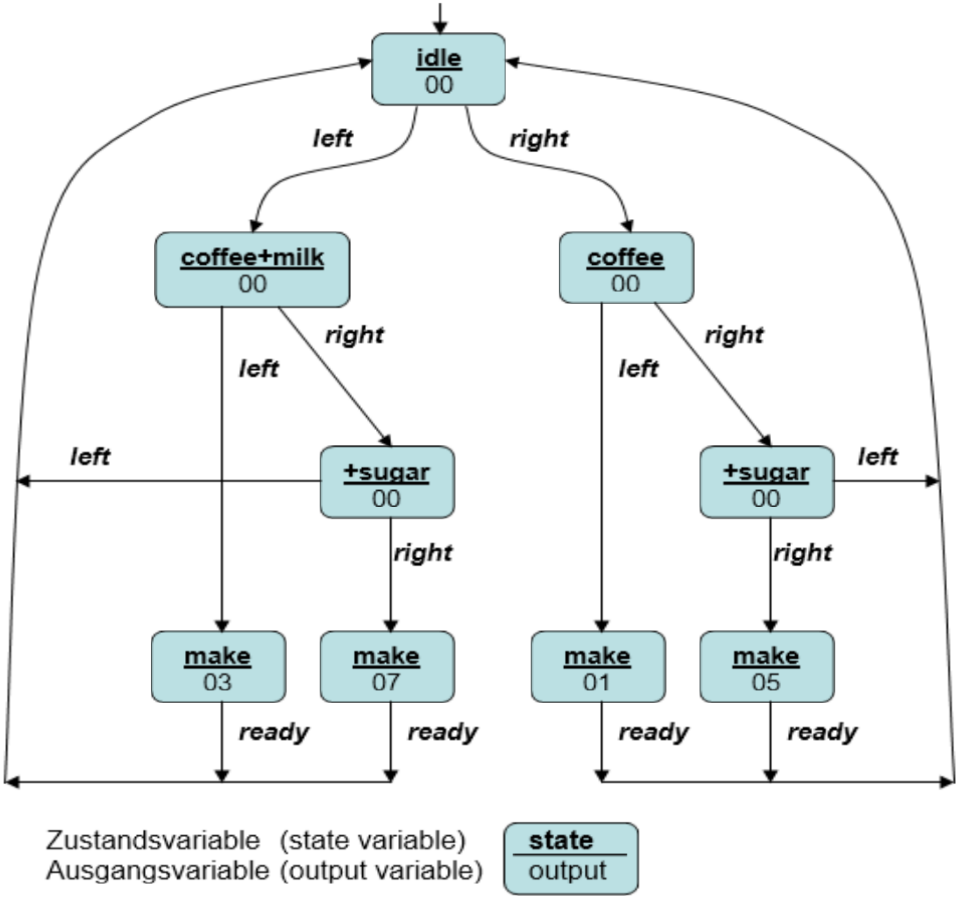
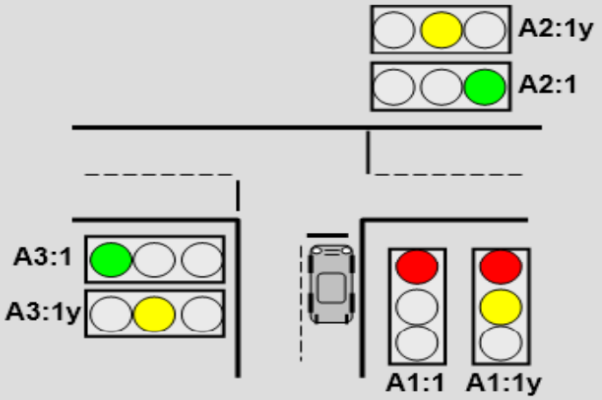


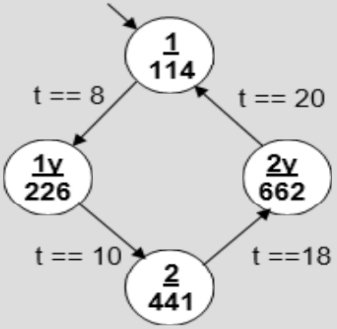
Figure 3.4.1: Finite State Machines (FSM)

state S	input	state S+	output
00 **idle (initial)**	*left* *right*	03: **coffee+milk** 01: **coffee**	**0** **0**
01 **coffee**	*left* *right*	05: **make coffee** 03: **coffee+sugar**	**0** **0**
02 **coffee+milk**	*left* *right*	06: **make coffee+milk** 04: **coffee+milk+sugar**	**0** **0**
03 **coffee+sugar**	*left* *right*	00: **idle** 07: **make coffee+sugar**	**0** **0**
04 **coffee+milk+sugar**	*left* *right*	00: **idle** 08: **make coffee+milk+sugar**	**0** **0**
05 **make coffee**	*!ready* *ready*	05: **make coffee** 00: **idle**	**01**
06 **make coffee+milk**	*!ready* *ready*	06: **make coffee+milk** 00: **idle**	**03**
07 **make coffee+sugar**	*!ready* *ready*	07: **make coffee+sugar** 00: **idle**	**05**
08 **make coffee+milk+sugar**	*!ready* *ready*	08: **make coffee+milk+sugar** 00: **idle**	**07**

Solution: Traffic Light as a Finite State Machine FSM, inputs: a ***timer-counter value***



State Diagram



Moore FSM State Machine: State Transients

Phase 1:	0	≤ t < 8 s	→ State 1 (initial)
Phase 1y:	8	≤ t < 10 s	→ State 1Y
Phase 2,	10	≤ t < 18 s	→ State 2
Phase 2y,	18	≤ t < 20 s	→ State 2Y

State 1:	A1 = red	A2 = A3 = green
State 1Y:	A1 = red+yellow	A2 = A3 = yellow
State 2:	A1 = green	A2 = A3 = red
State 2Y:	A1 = yellow	A2 = A3 = red+yellow

out (int16_t):

10 9 8	**6 5 4**	**2 1 0**	bit
r y g	r y g	r y g	Color
A3	A2	A1	