Real-Time Programming

1494

WS 2021/22

HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Contents

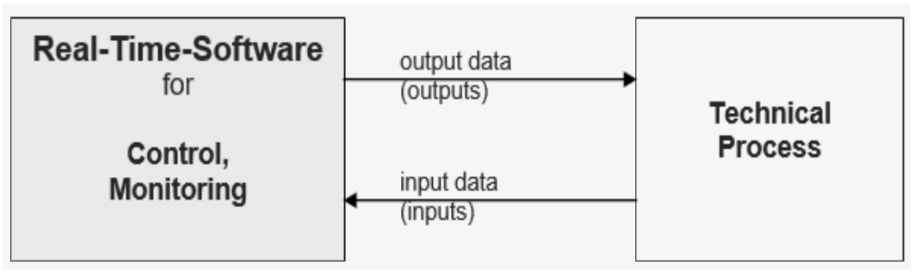
1 Real-Time Programming **3**
1.1 Real-Time Programming 3

Chapter 1

Real-Time Programming

1.1 Real-Time Programming

Real-time programming is about the acquisition and the processing of process variables (see section 1.1.1) of a technical process by means of digital programmable processors.

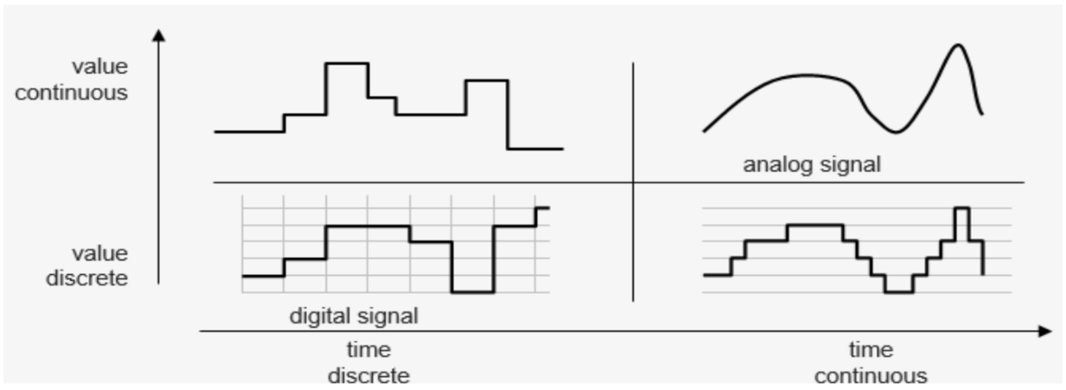


For the acquisition of physical quantities, there are **sensors**, which convert a physical quantity to be measured into an electrical signal (= a function of time).

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**.

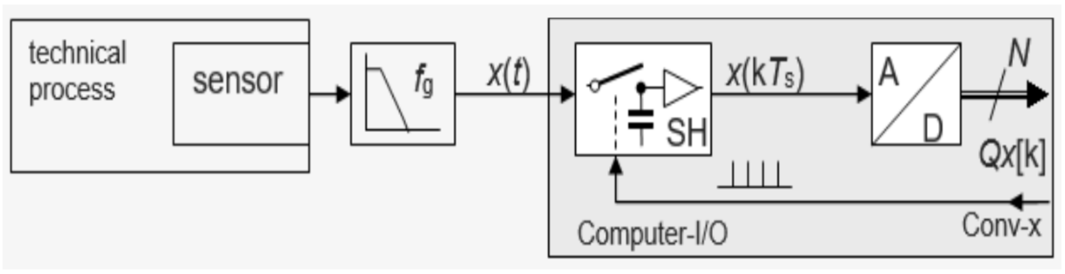


Since a process can be affected by actor (actuator), the **digital**, **time discrete** control signal, produced by a real-time computer program, must be converted back into an analog signal.

1.1.1 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 $f_g \leq 0.5 \cdot f_s$. 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 $x(\mathrm{k} \cdot T_s)$ are still value continuous (real valued), but values change only at discrete times $\mathrm{k} \cdot T_s$ (= **sampling**).

In a second step, the time discrete signal $x(\mathrm{k} \cdot T_s)$ 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

$x \approx Q_x \cdot c_x$ with c_x as some conversion factor (**??**).

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

1.1.2 Digital Sampling Systems

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

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.

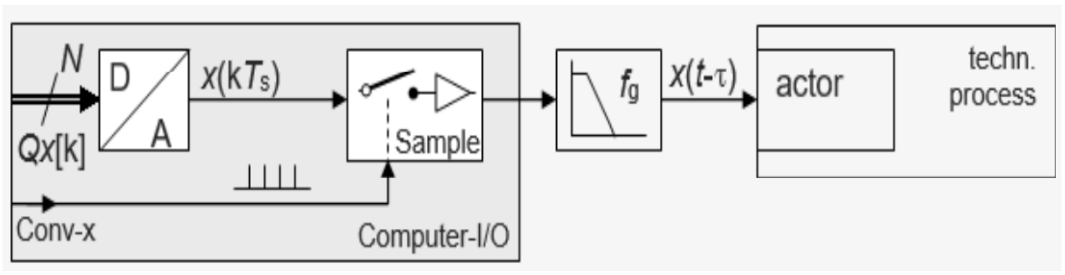
This is the fundamental theorem of digital signal processing.

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 $x(t$-), which is apart from a certain constant time delay the analog signal $x(t)$!

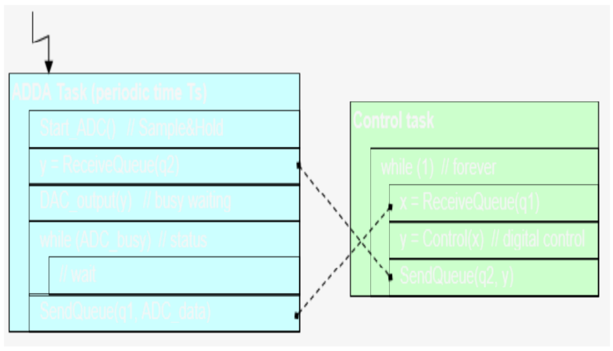
Real-Time Conditions for Sampling Systems

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%.

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

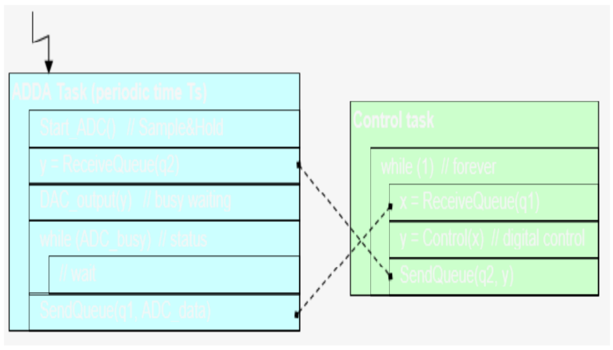

ADDA Task (periodic time Ts)
Start_ADC() // Sample&Hold
y = ReceiveQueue(q2)
DAC_output(y) // busy waiting
while (ADC_busy) // status
// wait
SendQueue(q1, ADC_data)
Control task
while (1) // forever
x = ReceiveQueue(q1)
y = Control(x) // digital control
SendQueue(q2, y)

*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 !!


ADDA Task (periodic time Ts)
Start_ADC() // Sample&Hold
y = ReceiveQueue(q2)
DAC_output(y) // busy waiting
while (ADC_busy) // status
// wait
SendQueue(q1, ADC_data)
Control task
while (1) // forever
x = ReceiveQueue(q1)
y = Control(x) // digital control
SendQueue(q2, y)

Quantization

A discrete-time signal $x(\mathrm{k}\cdot T_s)$ 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

$$\begin{aligned} x_Q &= Qx \cdot c_x + x_{\min} = Qx \cdot \tfrac{x_{\max}-x_{\min}}{N} + x_{\min} \\ c_x &= \tfrac{x_{\max}-x_{\min}}{N} = \tfrac{x_{\max}-x_{\min}}{2^M} \end{aligned} \tag{1.1}$$

Qx integer representation of x
xQ quantized physical quantity
cx conversion factor

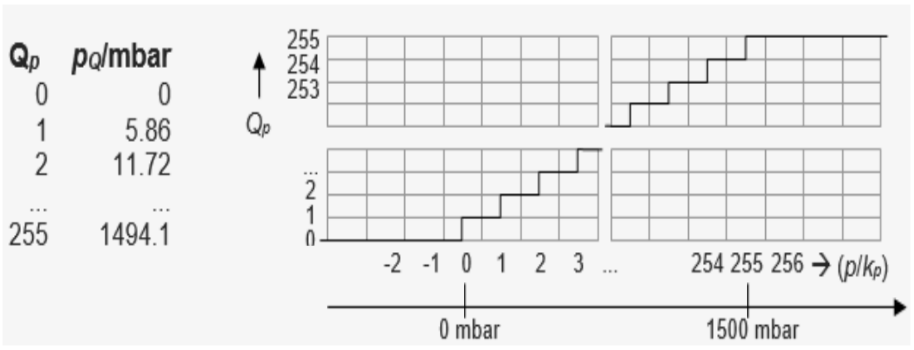
In automation 8-, 10-, 12-, 14- bit AD converters are used more often than, say, 16-, 20- or 24-bit ADCs, the latter are mainly used for audio signals.

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.



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!

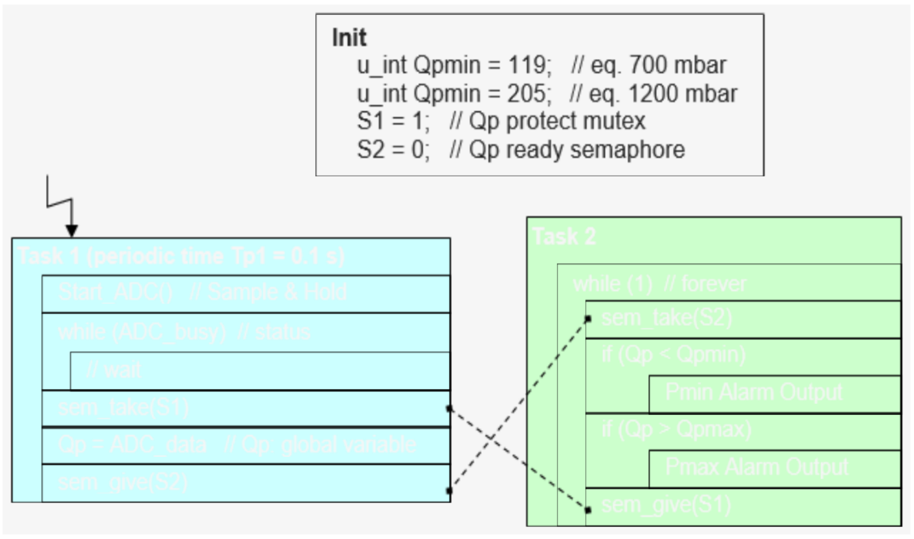
An appropriate real-time program for a microcontroller shall be designed with two tasks: 1. a task for pressure signal acquisition, 2. analysis and output.

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 Qp[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})



```
Init
    u_int Qpmin = 119;   // eq. 700 mbar
    u_int Qpmin = 205;   // eq. 1200 mbar
    S1 = 1;   // Qp protect mutex
    S2 = 0;   // Qp ready semaphore

Task 1 (periodic time Tp1 = 0.1 s)
    Start_ADC()   // Sample & Hold
    while (ADC_busy)  // status
        // wait
    sem_take(S1)
    Qp = ADC_data   // Qp: global variable
    sem_give(S2)

Task 2
    while (1)  // forever
        sem_take(S2)
        if (Qp < Qpmin)
            Pmin Alarm Output
        if (Qp > Qpmax)
            Pmax Alarm Output
        sem_give(S1)
```

1.1.3 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

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

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

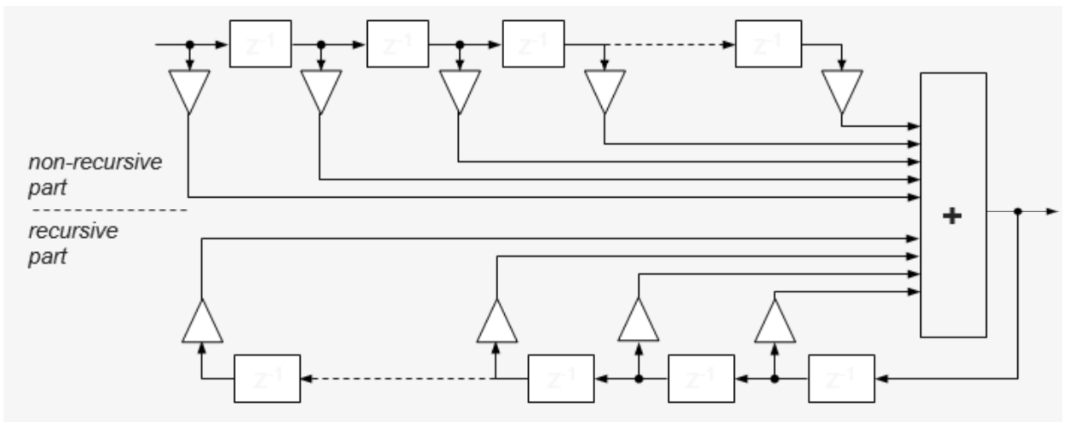
it is always possible to define $a_0 = 1$, which results in a recursion formula for the actual output value with constant coefficients a_j,b_j

$$y[n] = \sum_{j=0}^{M} b_j \cdot x[n-j] \quad - \sum_{j=1}^{N} a_j \cdot y[n-j] \; with \; a_0 = 1 \; (byconvention) \tag{1.2}$$

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):

$$y[n] = \frac{\sum_{j=0}^{M}(S\,b_j)\cdot x[n-j] \quad - \sum_{j=1}^{N}(S\,a_j)\cdot y[n-j]}{S} = \frac{A}{S}, \qquad S \in \mathbf{N} \tag{1.3}$$

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

$$\frac{A}{S}(A >> ldS) \qquad withoutroundop. \frac{A+S/2}{S}(A+S/2) >> ldS \qquad with\ round\ op. \tag{1.4}$$

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

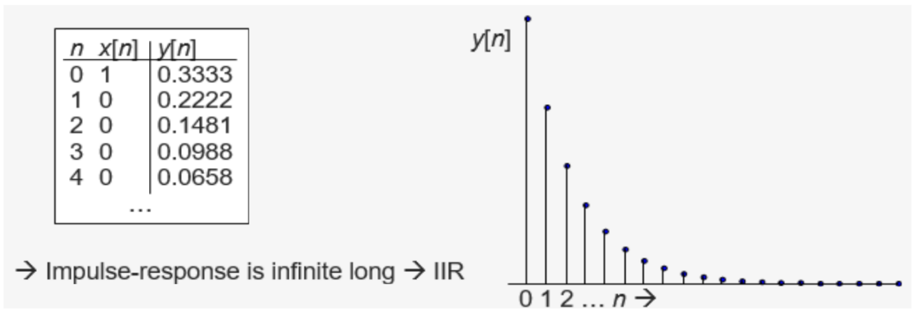
3. Avoiding of Modulo (%)
   Bitwise AND (&) with powers of two instead of modulo

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

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.

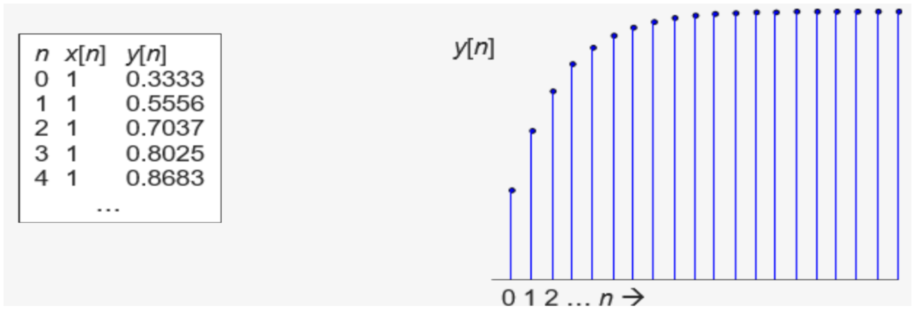
- **Impulse response** $x[n] = [n]$: $x =$ 1, 0, 0, 0, ...



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
	...	

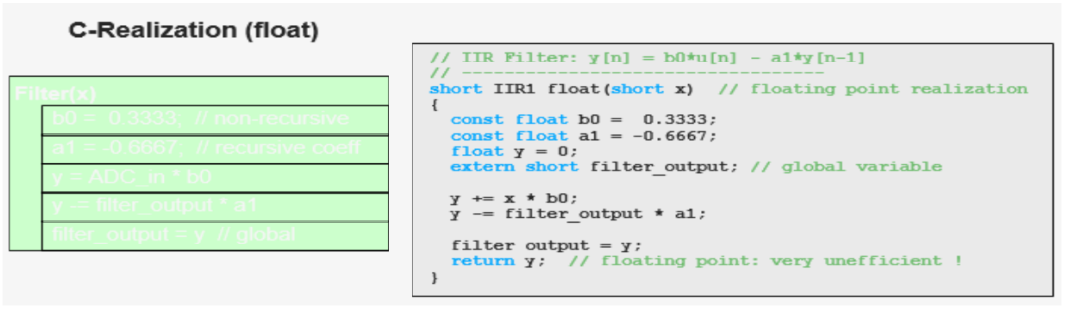
  Impulse-response is infinite long $\rightarrow$ IIR

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



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
	...	

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



C-Realization (float)
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]
// ------------------------------------
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 !
}
```

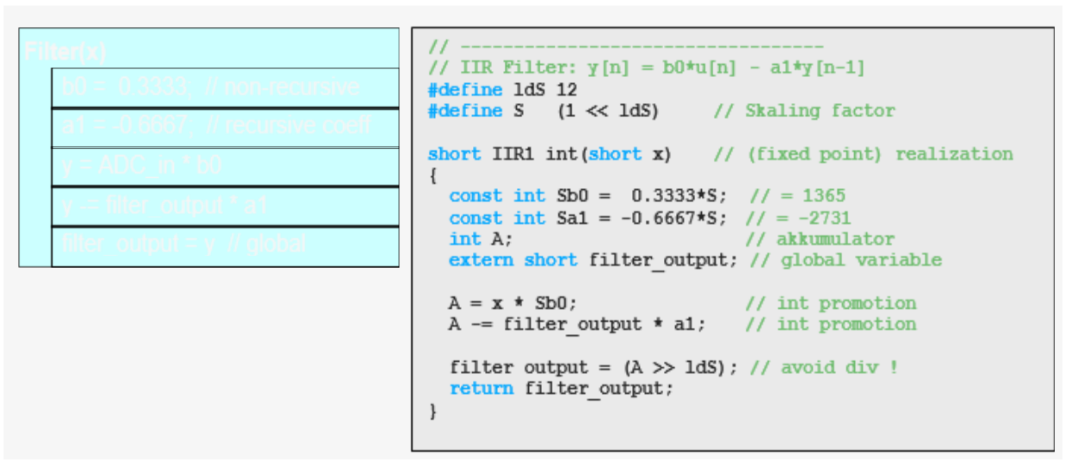
C- Realization (float): On microcontrollers without float-hardware needs hundreds of machine cycles and is thus very un-effective (section 3.2.1) !
C- Realization (integer): Using the integer difference equation (**??**) with a power of 2: $S = 2^{ldS}$

$$\begin{aligned} S \cdot y[n] &= (S\,b_0) \cdot x[n] - (S\,a_1) \cdot y[n-1] \\ S\,b_0 &= S \cdot 0.3333 \\ -S\,a_1 &= S \cdot 0.6667 \end{aligned}$$

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

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

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

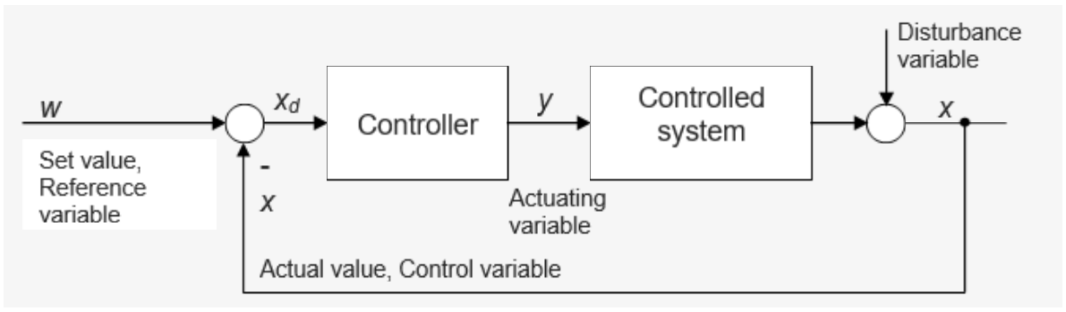
```
// ---------------------------------
// IIR Filter: y[n] = b0*u[n] - a1*y[n-1]
#define ldS 12
#define S   (1 << ldS)      // Skaling factor

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) !

1.1.4 Digital Controls

PID Controller

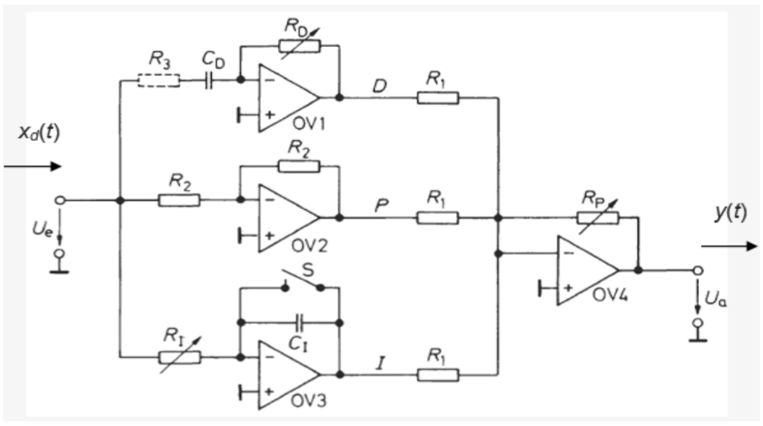
For many control systems in engineering, controls are used, which follow the so-called PID (**P**roportional-**I**ntegral-**D**ifferential) principle. Each of the 3 controller parts has its term with a specific parameter k_R, T_N, and T_V which allow the system response to be optimized with respect to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



System behavior of an analog PID Controller

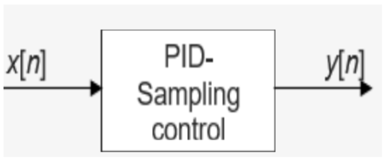
$$y(t) = K_R \cdot \left[x_d(t) + \frac{1}{T_N} \int x_d(t)\, dt + T_V \frac{dx_d(t)}{dt} \right] \tag{1.5}$$

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)

$$\begin{aligned} y[n] = {} & y[n-1] + \\ & K_R \cdot \left[x[n] \cdot \left(1 + \tfrac{T_V}{T_S}\right) - x[n-1] \cdot \left(1 - \tfrac{T_S}{T_N} + 2\tfrac{T_V}{T_S}\right) + x[n-2] \cdot \tfrac{T_V}{T_S} \right] \end{aligned} \tag{1.6}$$

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:

$$y[n] = y[n-1] + K_R \cdot \left[x_d[n] - x_d[n-1] \cdot \left(1 - \frac{T_S}{T_N}\right) \right] \tag{1.7}$$

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

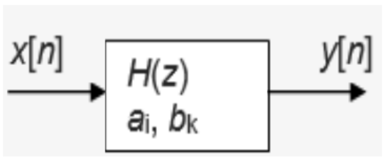
P-Controller

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

$$y[n] = K_R \cdot x_d[n] \tag{1.8}$$

PID-Controller as Digital Filter

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



$$[y[n] = \sum_{i=1}^{M} a_i \cdot y[n-i] + \sum_{k=0}^{N} b_k \cdot x[n-k]] \tag{1.9}$$

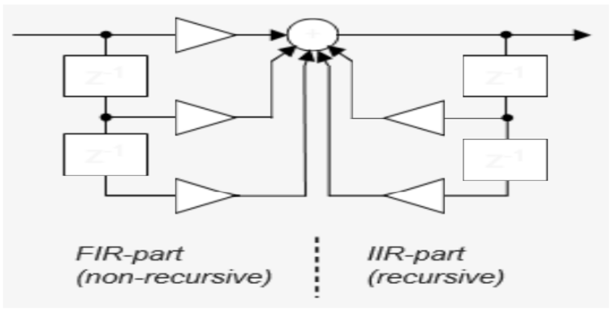
a common direct form (here, with input sequence $x_d[n] = x[n]$)

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.



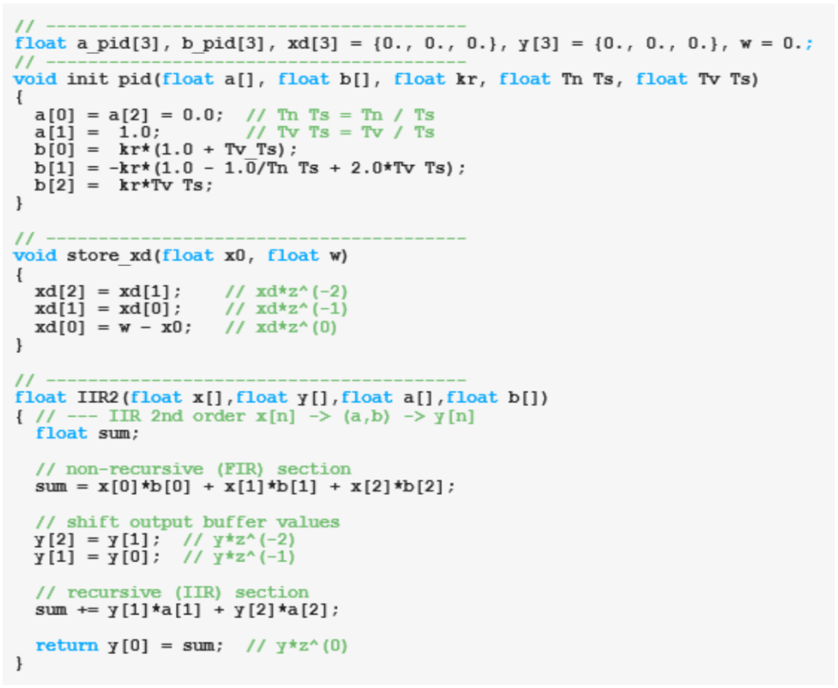
The algorithm (**??**) calculates a new output value $y[n]$, for each new sampling period T_S with a new input value $x[n]$ and two previous input values $x[n\text{-}1]$ and $x[n\text{-}2]$ and the last output value $y[n\text{-}1]$, which is stored.

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

- **IIR2**() realizes (**??**), 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 (**??**)



```
// ----------------------------------------
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).

- Events with period T_s and with **minimal** deviation (**jitter**) must be produced definition of a timer event (Ts)
- start of AD conversion for the **input** must occur immediately after the Ts-event, **minimal jitter**, (limited by the specified interrupt latency).

- The **output** value must be DA converted (with a delay of exactly one sampling period), with minimal jitter aas with AD conversion.
- It is important to ensure that the algorithm operates only with a **consistent set** of parameters and input and output buffers with consistent values.
- 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 $100 \cdot T_s$.It must be ensured, that any parameter change, always results with a stable digital filter.
- The set value must be adjustable from outside (e.g. by asynchronous bus event). The latency time allowed here is $20 \cdot T_s$.
- 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

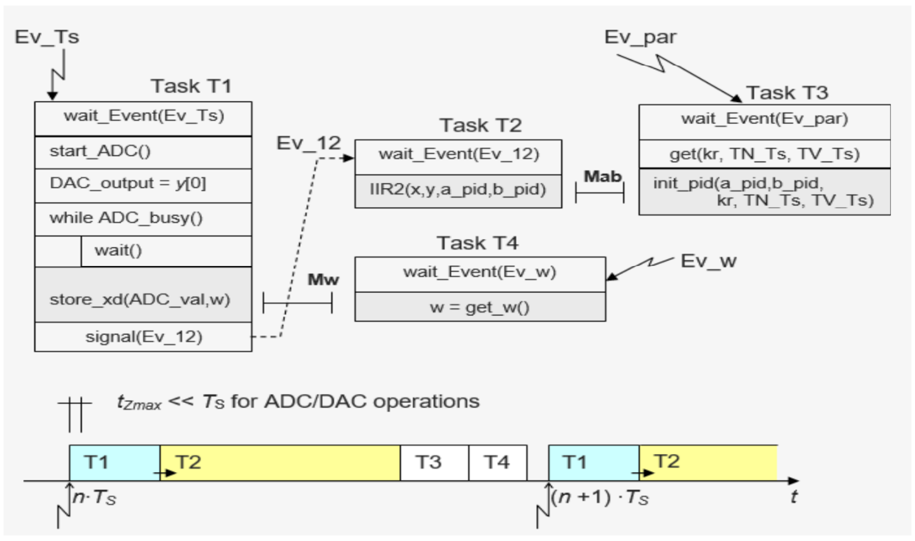
- **w (set value)**: read by T1 and can be concurrently written by T4 Mutex **Mw**
- **a_pid, b_pid PID filter coefficients**: These are calculated and read by T2, concurrently written by T3 Mutex **Mab**

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

1. 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.
2. by **Ev_T12** is ensured, that T2 runs only when T1 is finished (cooperation sync)
3. by Mutex **Mab is** ensured, that the PID algorithm is executed only with a consistent set of parameters.
4. 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)


Ev_Ts
Task T1
wait_Event(Ev_Ts)
start_ADC()
DAC_output = y[0]
while ADC_busy()
wait()
store_xd(ADC_val,w)
signal(Ev_12)
Ev_12
Task T2
wait_Event(Ev_12)
IIR2(x,y,a_pid,b_pid)
Mab
Ev_par
Task T3
wait_Event(Ev_par)
get(kr, TN_Ts, TV_Ts)
init_pid(a_pid,b_pid, kr, TN_Ts, TV_Ts)
Task T4
wait_Event(Ev_w)
w = get_w()
Ev_w
Mw
$t_{Zmax} << T_S$ for ADC/DAC operations
T1
T2
T3
T4
T1
T2
$n \cdot T_S$
$(n+1) \cdot T_S$
t