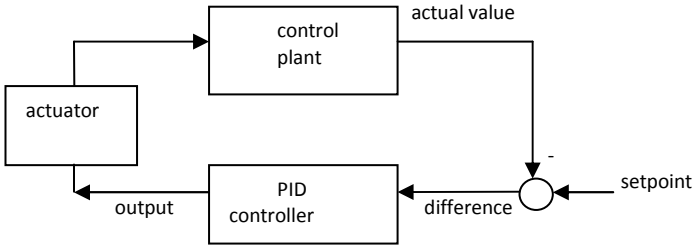
Library Modular PID controllers

Fundamentals in few words.

How the PID controllers work.

In internet many pages tell about control systems and PID controllers,
e.g. in Wikipediahttps://en.wikipedia.org/wiki/PID_controller
At the beginning of 20-th century the PID controllers were invented and
remain till today in all most cases the best solution, rarely different kind
of controllers are necessary.
Theoretically no a better way of stabilizing a physical value in closed loop
as using PID algorithm exists. Any others attempts, except same rare cases,
have brought not much improvement, but have done much complication.
In the control science we use a concept of plant.
In this abstract concept we consider an output of the plant - some measured value,
which we want to control (almost every time stabilize)
and input of the plant, which is a physical value we are able to change
through the action of controller (actuator).



Control plant and controller

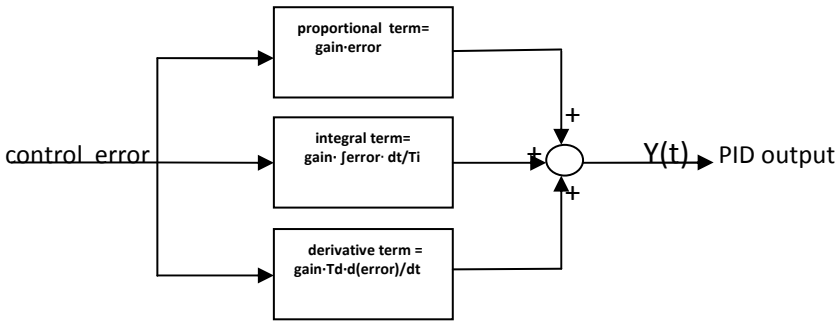
We are not interested in physical construction of some apparatus, but want
to see the response of the plant, it means the reaction of the measured
value, after a change of input value.
See https://en.wikipedia.org/wiki/Step_response
PID controllers have got three main parameters:

gain
Ti integration time
Td derivative time

and calculates the following output, comparing setpoint to actual value or
directly **error(t) = setpoint(t) - actual value(t)**

Y(t) = gain(error + ∫error·dt/Ti + Td·d(error)/dt)

This output of the controllers regulates feedback of the plant causing
the actual value remaining stable.



PID controller

This time function Y(t) conducts to Laplace transformation of transit function:

K(p) = gain(1 + 1/pTi + pTd)

where p is the operator replacing differentiation und 1/p replacing the integration.

Calculation of controller output Y(t) in program

For **Y(t)** calculation we need in Arduino integration and differentiation.
Principle of integration and differentiation is to do calculations by constant time difference between two calculation steps.
We speak about constant sample time. Only in this way the continuous differentiation and integration could be done properly in a discreet way.
The constant sample time is realized by control functions **Takt100** and **Takt1000.**
The sample time must be short enough to control fast changing actual values.
For almost all practical cases of control plant 100 ms created by **Takt100,** the standard Library PIDcontrollersModular sample time, is sufficient.

Tuning of controllers in few words

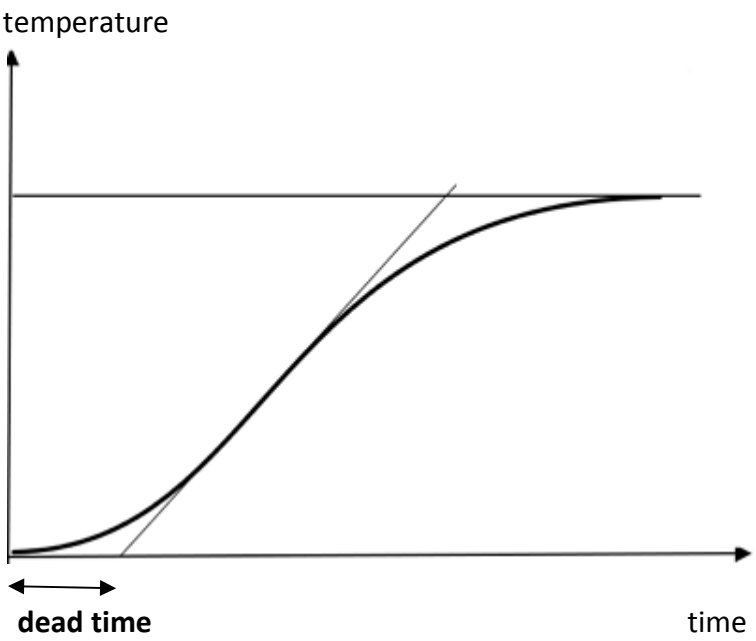
After some jumping change of input of the plant we get a time response of the output, which contains all dynamic information about the plant.

see https://en.wikipedia.org/wiki/Step_response

This response is described using Laplace transformation.
The sophisticated mathematics must not be known for a common user of controllers, but simply rules of thumbs are enough to tune the controllers.

Static plants

In almost all cases plants are static; it means, after change of input, the output after some time gets some stable value, e.g. after heating on, the temperature begins to rise and reaches some stable value.



Response of a static plant : temperature to heating on

For the tuning of controllers we need to know the dead time of the response.
Simply use your stop watch to look for the first rise of temperature, after heating on. You will get the dead time of the static plant.
And then calculate the following control parameters:

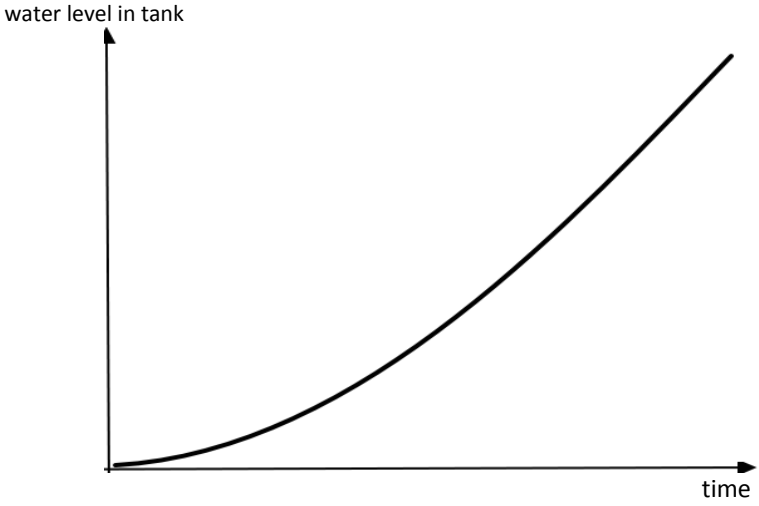
derivative time = dead time/2
integration time = 4 * derivative time

Gain put at the beginning =1.0 and increase the gain looking at control loop behavior, if is stable and its overshoot is not too big.
If the gain is too big, control loop oscillates.
This oversimplified way of tuning was first found by Ziegler-Nichols.
Original Ziegler-Nichols method is far more complex and is described in details on the internet, e.g. in wikipedia:

https://en.wikipedia.org/wiki/Ziegler%E2%80%93Nichols_method

Astatic plants

In this kind of plants some equilibrium state is reached between input and output in control loop, but it is fragile.
Every small change of input causes higher or lower running of the output.
You can imagine a ball placed on some top, which go down after some little touch.



Response of an astatic plant. Water level in tank
- pump in outlet was switched off.

Control of water level in a tank could serve as an example of an astatic plant.

There are two pumps.
One pump is supplying water to the tank, which is flowing outside
by action of another pump on output.
If one pump changes flow and another pump flow remains, the level in the tank goes up or down continuously.
The introduced before simple thumb rules cannot be used in this example, because this plant is astatic.
Look into control books for other formulas.
Generally the controller integration time interact with plant integration causing instability in system.
Be careful using Ti, if too small, your system could be instable.

Actual value of controllers in Arduino

The continuous controller requires floating value in the range [0 -100], the Arduino analog value is type int 0 - 1023 and the following commands satisfy the demand**:**

```
int value;
value=analogRead() ;
act=value*/10.23 ;
```

where **: act** controller actual value

In order to control a physical value we measure it and connect voltage signal

from the measurement instrument to Arduino 0 – 5V analog input and
read this value in program using **analogRead( )** command.
We also need in setup **analogReference( )** command to define reference voltage.
How to do this exactly please look into Arduino reference or manual.
As result we get 0 -1023 integer value.
Naturally the setpoint of controller has to be also in the range of 0 -100 floating value.
Controllers can also obtain actual value in many other different ways, e.g using bus transmission or any kind of interface to Arduino.
Two points are important:

1. Range and linearity of signals
2. Sampling time of signals and sampling time of controllers

1. **range**

These introduced controllers need a linear physical actual value, and then the setpoint has to be also linear, in the range 0 – 100. Such linear actual value can be easily indicated.
In rarely cases of nonlinearity the control function **Linear** is useful.

2. **sample time**

If the control loop is fast – e.g. control of motor speed, all signal and components must be able to transmit the information in the control loop, we speak about the limit frequency and sample time.
Not to indulge into the sophisticated mathematics let us say, controllers should run in a short cycle in Arduino, analog values must be processed in a short time and supplied fast.
If not, the control loop can't react quickly to disturbances in loop, therefore, it is not able to control properly actual value.
These controllers are written for the standard sample time 100 ms, in many instances short enough, but can run in actually any different sample time, e.g. 10 ms or 1 s.
But then time dependent controller parameters should be calculated anew.
E.g. controllers running in 100ms sample time (**T100**) was tuned to Ti=100s, Td=25s should run now in 1 s cycle(**T1000**) , then:

Ti new = Ti /10 =10s
Td new = Td/10 =2.5s

The same conversion affects all other control function, which are processed in time.
Their time depended parameter need to be corrected like in previous example.
One thing is important – the sample time must remain constant, otherwise calculations of PID algorithm will flow in time, creating new difficult problems in control system.
Small changes of controllers' cycle don't matter much.

Takt1000 and **Takt100** will create constant sample time with enough accuracy,
if only Arduino loop is not too heavily loaded doing processing (fast enough).

Output of controllers in Arduino

Arduino analog output requires integer values in the range [0 -255] and the following commands satisfy the demand**:**

```
int value;
value=out*2.55;
analogWrite( value );
```

where **: out** controller output

Controller **CPID** generates continuous output which should be converted into some electric analog signal, e.g. current or voltage before going to actuator.
There are no analog output cards by Arduino.
Although we use command **analogWrite**, but the analog values are converted into digital modulated output, by way of pulse width modulation (PWM) and should be linked to a digital output pin configured with **pinMode**.
Look into Arduino reference and manuals, how it is functioning.

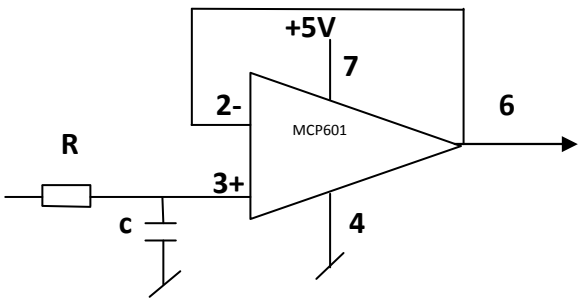
please note:
Only pins 3,5,6,9,10,11 are available as analog output.

PWM frequency on pin 3,9,10,11- 490 Hz
PWM frequency on pin 5,6 - 980 Hz

This frequency can be changed using command:

analogWrite(pin, pulsweite)

In the simplest way, this signal can be transformed into voltage
using low pass filter: capacitor c=10 µF , resistor R= 4.7 KΩ, operational amplifier rail-to-rail e.g. MCP601. Unfortunately the accuracy is not the best, below 1 %.
Much better is to implement a special chip LTC2644.



Low pass filter for Arduino analog output

Low pass filter is worsening the controller output dynamic, but that is
only relevant in very fast control loops.
Two remaining controllers **SPID, CPID,** produce digital outputs which should be
simply connected to Arduino digital outputs using Arduino command **digitalWrite**().
Before using **digitalWrite()** the pin must be configured to output
e.g**.** pin 6 define as digital output, **pinMode**(6, OUTPUT) (see examples)

Arduino digital inputs

Be careful using Arduino digital inputs, because they have very high input resistance
>100 MΩ and therefore are very sensitive to noise from the environment.
Use external resistors or Arduino command **INPUT_PULLUP** (internal resistor).
Read Arduino reference or manuals.
Inputs 0,1 are used for serial communication and are not available for programs
in **examples** of this library.
They may be implemented, when you erase the communication
to plotter, it means all commands **Serial.begin**(9600) initializing the communication.

About the library PID controllers general.

This library contains a scope of 6 files cpp+header with PID controllers and two
files cpp+header with control functions, which are useful for control systems.
The library is composed modular and very flexible: either one controller
or many controllers could run simultaneously.
Control functions could also be implemented for different purposes outside
control systems.
Call in your program only controllers and functions you need and connect
their inputs with outputs in order to get desirable performance (see examples)
and call in a proper sampling time using **T100** and **T1000**.
There are 3 different types of controllers and 12 different control functions:

controllers

class **CPID** continuous PID controller
class **SPID** step PID controller for servomotors
class **IPID** impulse PID controller with impulse-pause modulation (PWD) for heating and cooling.
call method **cupdate**.

control functions

time sampling and shifting of calls:
class **Takt100** 100mssampling
class **Takt1000** 1s sampling

actual value preparing:
class **Mean** calculating mean value
class **PT** low pass filter
class **Diff** high pass filter
class **Dead** dead time
class **Int** integration
class **Linear** linearizing
class **Hyst** hysteresis between actual value and setpoint

setpoint preparing:
class **SetGen** increase or decrease setpoint in steps, after push button is pressed.
class **SetJpRamp** replace set point jump through ramping
class **SetTimeTable** generate setpoints according to values defined in time table

call method **fupdate**

In this library every type of controller has a separate cpp file.
Additional control functions are collected in **ConFunct** file.
These control functions mostly prepare actual value and setpoint of
a controller improving its performance in a control loop.
Generally controllers don't like bumping and disturbances of actual value and setpoint, but prefer smooth signals. Then they produce a smooth output, which is better for actuators and create no worsening of control loop performance, but improving.

Control functions:

1. Filtering of disturbances in measured signal
   The filtering, low pass filter **PT** und mean value **Mean** , has to be done carefully, not too strong, not to eliminate the necessary dynamic of the control loop.

2. Eliminating nonlinearities in controlled loop, especially in measured signal.

3. Providing setpoint in a smooth way – avoiding jumping of setpoint, and providing setpoint according to a stored tables.

All controllers need a constant sample time, because PID algorithm desires an exact calculation of time. Laplace functions must be sampled by constant time, which can only be done with some limited accuracy in loop, because there are no standard hardware time interrupts in Arduino.

Programs in folder examples

Half of these examples contains applications of controllers, another half simulation loops.

About the applications

In the **examples** there are three files with application of **CPIDApplication**, **SPID Application** and **IPIDApplication** and one with controllers cascade in file **CPIDApplicationCascade.**
Every program has the same construction.

1. Configure Arduino inputs and outputs.
2. Read actual value using command **analogRead**( ) and filter this value using **PT** control function T=2s in order to eliminate noise. Define setpoint **set**=50.
3. Connect bits switching auto/manual and push buttons bits for controller output manipulation in manual mode to peripheries.
4. Put controllers output to peripheries. Depending on controller output either analog value **CPID** or digital outputs – **SPID, IPID**.

Controller cascade in file CPIDApplicationCascade

Two continuous controllers **con1** master controller and **con2** follow up controller are running. Actual values are supplied through commands **analogRead**() and filtered using low pass filter **PT** with time constant 2s in order to eliminate noise in analog signal.
Both controllers get the same digital signal switching auto/manual.
In manual mode the output of the follow up controller can be manipulated by pushing buttons up and down. The output of master controller goes as setpoint to follow up controller. Control parameters of follow up controller are tuned expecting fast loop, master controller is prepared for slow loop.

About the simulation

All three types of controllers are built in an identical control loop and get to execute the same task: control actual value accordingly to ramp generated by **SetTimeTable** and **SetJpRamp.**
In all three cases the control plant is the same:
PT element (low pass filter) with T=100s connected to dead time element **Dead T** = 10s.
In such a case controller parameters, according to Ziegler-Nichols, are approximately as follows:

Ti= 2* Dead T= 20s
Td = 0.5 *Dead T = 5s.

The gain was tuned experimentally: gain=6.
You can watch behavior of the controllers, compare their work, change parameter and examine their influence on the control loop quality.
Please use the Arduino plotter to watch the simulation.
In case of **IPID** and **SPID** extra function were used to produce analog actual value from digital outputs of these controllers:

Step controller SPID

SPID digital outputs are connected to integrator **Int** in order to simulate the work of servo motor:

put analog value:
100 if impulse motor direction +
0 no impulse
-100 if impulse motor direction -

Impulse controller IPID

IPID outputs are converted to analog values and lead directly as input
for **PT** element: 100, 0, -100

put analog value:
100 if heating on
0 no heating, no cooling
-100 if cooling on

please note
Not every pulse on **SPID** and **IPID** controller output appears in simulation on plotter, because they are sometimes too fast, too short and the plotter can't depict them, but they are not lost and show up on digital outputs.

Continuous PID controller CPID

Call every 100 ms using function **Takt100.**
Continues controllers are used generally to stabilize physical values
on a plant. Many different applications are possible and could be found in
control system literature.
The output signal of a plant, which should be hold constant, is measured
and compared to the setpoint.

*The introduced **CPID** controller is also able to work together*
*with a second controller in cascade (see **examples** CPID cascade).*

Continuous PID controller calculates the error difference between
the actual value und desired setpoint and generates a continuous output
as a sum of proportional integral and derivative term, which goes to the
actuator , according to the following formula:

Y(t) = gain (error + ∫(error.dt)/Ti + Td*d(error)/dt)

controller parameters:

float **act** [0 -100] actual value
float **set**[0 –100] setpoint
float **diff** [-100, 100] control difference
float **out** [0 – 100] PID output value
float **track** [0 – 100] tracking value , only for cascade of two controllers
bool **auto_man** switching auto/manual
bool **out_up** push button output up
bool **out_down** push button output down
bool **cascade** if cascade=true, master controller

float **gain** controller gain
float **Ti**[s] controller integration time
float **Td**[s] controller derivative time

internal data:
float **P** proportional term
float **I** integral term
float **D** derivative term

operation modes

Two modes are possible.

manual mode: auto_man = 0

It is necessary to switch into the manual mode during setup of plant, in cases of trouble or simply to test the control loop.
And it is almost impossible to run the controller only in auto mode.
In manual mode output of the controller can be manipulated using
bits **out_up** und **out_up**. These bits can be linked directly to hardware push buttons.

out_up output of the controller higher, push button +
out_down output of the controller lower, push button -

If button pressed, the continuous output runs in the range from 0 to 100 with the speed 1% per second, higher or lower.
In this mode is possible to write some value direct to the **out** parameter, which the controller accepts as new starting point.
Switching manual/auto happens bumping free (no jump of output), if the control difference is zero.

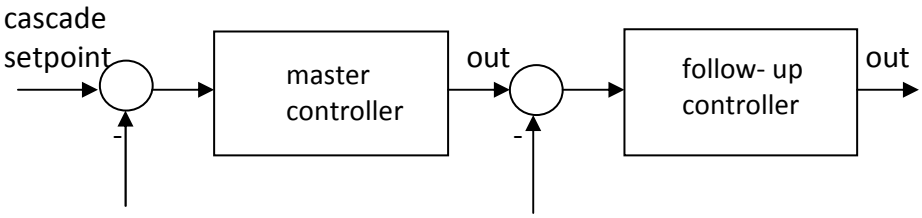
automatic mode: auto_man = 1

In this mode the output of the controllers is calculated accordingly
to PID formula:

Y(t) = gain (error + ∫(error.dt)/Ti + Td*d(error)/dt)

cascade

The cascade is built of two controllers: the master controller output goes as setpoint to follow up controller. The follow up controller works in
a fast control loop, the master controller loop is slow.



Cascade of controllers

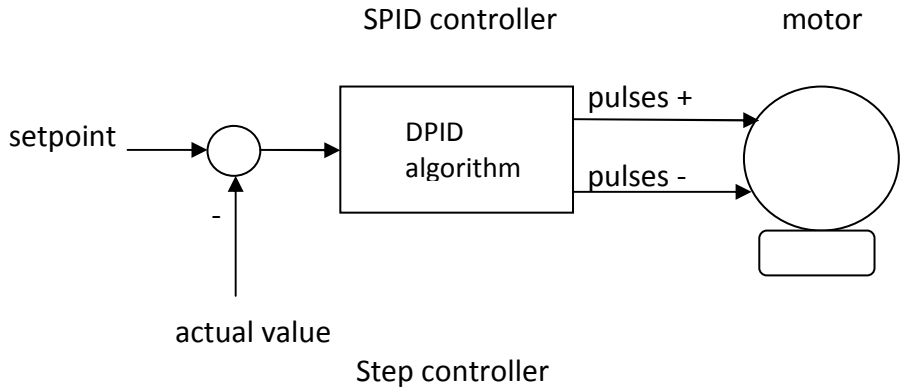
Please read about cascade in Wikipedia:
http://en.wikipedia.org/wiki/PID_controller , cascade control
or e.g.:
https://www.west-cs.com/news/how-does-cascade-control-work/

In single mode switching manual/auto should happen bumping-free, it means, without jumps of controller output. Controller should begin its work softly from last output, when the control difference is zero.
In case of follow-up controller in cascade the output of master controller serves as setpoint in the follow-up controller. To get rid of the follow up controller bumping, the analog input **track** in master controller gets the actual value of the follow-up controller and after the switching to auto produces output equals actual value which is put to the follow-up controller setpoint. The control difference is zero and no bumping occurs.
Write cascade =1 in master controller parameter and connect parameter **track** to follow-up controller actual value **act** (see examples, **CPIDApplicationCascade**).

initialization
In Arduino setup loop the **CPID** controller is initialized automatically.
Write some value in setup loop to parameter **out**, it will be taken as starting point if actual value equals setpoint during setup. Default value is zero.

Step PID controller SPID

Call every 100 ms using function **Takt100**.
This controller generates impulses for servo motor: move direction plus or minus.
The movement is equivalent to integration where output of the integrator is the motor rotor position.
Inside the program so called speed DPID algorithm is calculated and after the motor integration becomes PID algorithm like by continuous controller.



controller parameters:

float **act** [0 -100] actual value
float **set** [0 -100] setpoint
float **diff** [-100, 100] control difference

bool **auto_man** switching auto/manual
bool **butt_plu** push button direction plus
bool **butt_min** push button direction minus
bool **motor_plu** digital output direction plus
bool **motor_min** digital output direction minus

bool **limit_plu** limit switch direction plus– stopping pulses and DPID calculation
bool **limit_min** limit switch direction minus – stopping pulses and DPID calculation

float **gain** controller gain
float **Ti**[s] controller integral time
float **Td**[s] controller derivate time
float **onvalue** [0 – 100]
float **offvalue** [0 – 100]
float **mimp**[s] minimal motor impulse length
float **motime**[s**]** motor running time from minimal to maximal position

internal data:

float **PxD** speed proportional term
float **I xD** speed integral term
float **DxD** speed derivative term

operation modes

There are two operation modes: manual and auto.

manual mode: auto_man = 0

It is necessary to switch into the manual mode during setup of plant,
in cases of trouble, or simply to test the control loop.
And it is almost impossible to run the controller only in auto mode.
In manual mode output of the controller can be manipulated.
Pressing the push button **butt_plu** causes motor rotating direction plus,
the push button **butt_min** rotation direction minus.

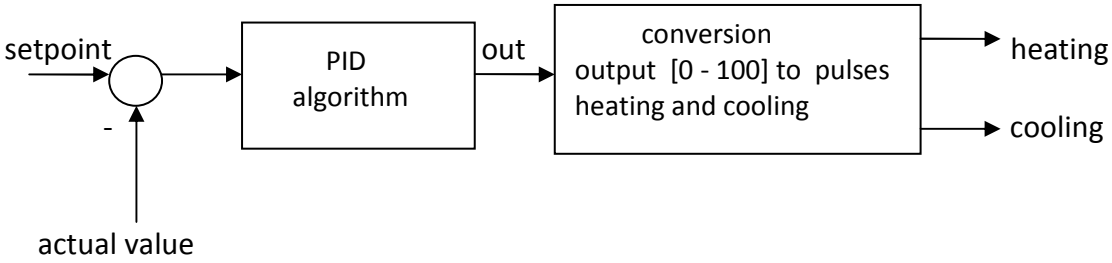
automatic mode: auto_man = 1

DPID speed algorithm is calculated and put out in form of pulses on
digital outputs **motor_plu** and **motor_min**.
If limit position is reached and one of limit switches is activated,
the generation of pulses is stopped.
Without limit switches processing of **DPID** will be limited inside, but
pulses are not stopped.
Limit switches do this better, stopping also pulses in right moment
and therefore are recommended.
Parameter **mimp** prevents too short pulses that could damage motor.
Near the point **diff**=0 the action of motor could be limited by means of
built in hysteresis with parameters **onvalue** and **offvalue**.
How it happens is described in case of control function **Hyst**.
These two parameters cannot be too big because of control loop accuracy.

initialization
In Arduino setup loop the controller is initialized automatically.

Impulse PID controller IPID – temperature controller

Call every 100 ms using function **Takt100**.



Impulse controller

controller parameters:

float **act** [0 -100] actual value
float **set** [0 -100] setpoint
float **diff** [-100, 100] control difference
float **out** [0 -100] continous output
float **track** [0 -100] tracking value, only for cascade

bool **auto_man** switching auto/manual
bool **butt_pl** push button direction plus
bool **butt_min** push button direction minus
bool **heating** output heating
bool **cooling** output cooling

bool **three_two** controller type =false, only heating, =true heating and cooling
bool **cascade** if cascade =true, master controller

float **gain** controller gain
float **Ti**[s] controller integration time
float **Td**[s] controller derivate time
float **period**[s] impulse/pause period by PWD modulation
float **min_imp**[s] minimum impulse length
float **cool_to_heat** ratio cooling/heating in the range < 0, 1 >
float **zone_high** [0-100] control zone high
float **zone_low** [0 -100] control zone low

internal data:
float **P** proportional term
float **I** integral term
float **D** derivative term

In this controller output of a continuous controller [0 -100] is converted into heating/cooling pulses by impulse/pause modulation (PWD).
The period is constant and defined in seconds by parameter **period**.

Types of controller

Two types of controllers are possible: parameter **three_two** = 0 only heating, **three_two** = 1 heating and cooling.

three_two = 0 controller output 0 – 100 heating pulses 0 – 100.
three_two = 1 controller output 0 - 100 cooling 0 -50, heating 50 - 100

Two short pulses (pauses) are eliminated through choice of minimal impulse length, parameter **min_imp** in seconds

operation modes

There are two operation modes: manual and auto.

manual mode: **auto_man** = 0

It is necessary to switch into the manual mode during setup of plant, in cases of trouble, or simply to test the control loop.
And it is almost impossible to run the controller only in auto mode.
In this mode the controller output **out** can be manipulated in the range [0 – 100] using the push buttons **butt_pl** direction up with a speed 1% per second and **butt_min** direction down with a speed 1% per second.
As result of the controller output manipulation, length of heating (and cooling pulses) pulses changes from value zero to period.

automatic mode: **auto_man** = 1

PID algorithm is calculated and put out in a form of heating and cooling pulses on digital outputs **heating** and **cooling**.

Parameter **min_imp** prevents too short pulses that could damage the heating (cooling) units.
Parameter **cool _to heat** defines the ratio cooling/heating and is smaller 1.
If **cool_to_heat** =1, then by analog output **out**=0 cooling impulse length equals period (full cooling).
If **cool_to_heat**=0.5, then by **out**=0 cooling pulse length equals 50% of period.

Zones **zone_high** and **zone_low** are switching action of PID algorithm off

in the following way:

1.
actual value temperature overshoots **act >=set + zone_high**
then heating (cooling) permanent off, PID integration is stopped.
2.
actual value temperature goes down **act <=set - zone_low**
then heating permanent on (cooling off), PID integration is stopped.

initialization

In Arduino setup loop the controller is initialized automatically.
Write some value in setup loop to parameter **out**, it will be taken as starting point, if actual value equals setpoint during setup. Default value is zero.

List of programs in ConFunc file.

Takt100 and Takt1000

The fundamental classes in this library are **Takt1000** und **Takt100**.
They create time sampling 1s und 100 ms and in the same time they share calls of controllers und control functions.

Takt100 returns values: 0,1,2,3,4 every 100 ms shifted at 20 ms one after another.
Takt1000 returns values 0,1,2,3,4 every 1 s shifted at 200ms. one after another

Both return 5 in case of no time sampling.

They cannot do this perfectly, but with the accuracy of loop time.
They use for this task system function **millis()**.
Although this way is not perfect, it brings enough accuracy in most cases.

Using of Takt1000 and Takt100.
Only one instance of **Takt1000** and **Takt100** should be called in loop, because only then we get their desired performance: not only the right sampling, but also the right microprocessor time sharing.

The time sharing.
If many controllers are called in the same cycle, they create too big burden for the Arduino microprocessor.
Functions **Takt1000** and **Takt100** not only create the sampling 1000ms and 100ms, but also share the burden of processing into many Arduino cycles.

How to call **Takt100** and **Takt1000**.

example for **Takt100** and continuous controller **CPID** (look into Arduino examples)

```
loop (
value=takt100.fupdate();
if (value ==0 )
{ con1.cupdate(); }     //instance 1 of controller is processed every 100 ms
if (value ==1)
{ con2.cupdate(); }     //instance 2 of controller is processed 20ms later, every 100 ms

.................
.................

 if (value==4)
{ con5.cupdate();}
 )
```

where:

takt100.fupdate() // call method **fupdate** of class **Takt100**
con1.update( ) // call method **update** of class **CPID**

Function Mean

Call in 100 ms cycle using **Takt100** sample time or **Takt1000**1 second sample time.

parameters
float **in** signal input
float **out** output mean value
bool **start** if start=true initialize

Calculate a mean value of last 8 sampled values according to formula:

out = (in + 7 out last value)/8

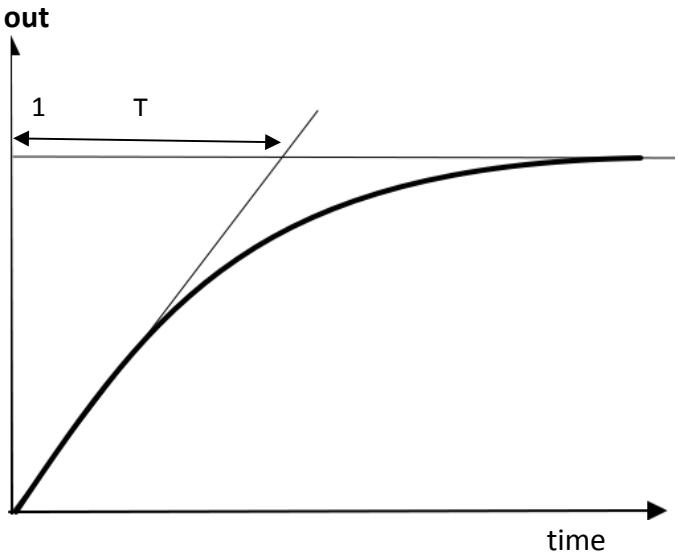
Be careful, on this way signal loses noise and disturbances, but also some dynamic can be lost, which is necessary for good performance of control loop.

initialization
In Arduino setup loop the controller is initialized automatically, if bit **start**=1.
Put **start**= 1 to initialize.

Function PT

Call every 100 ms using function **Takt100**

parameters
float **in** input signal
float **out** filtered output
float **T**[s] time constant



Response of PT element to step **in** jump =1

Function **PT** is a low pass filter, suitable especially for the filtering the actual value, realizing the following Laplace function:

1/1+pT

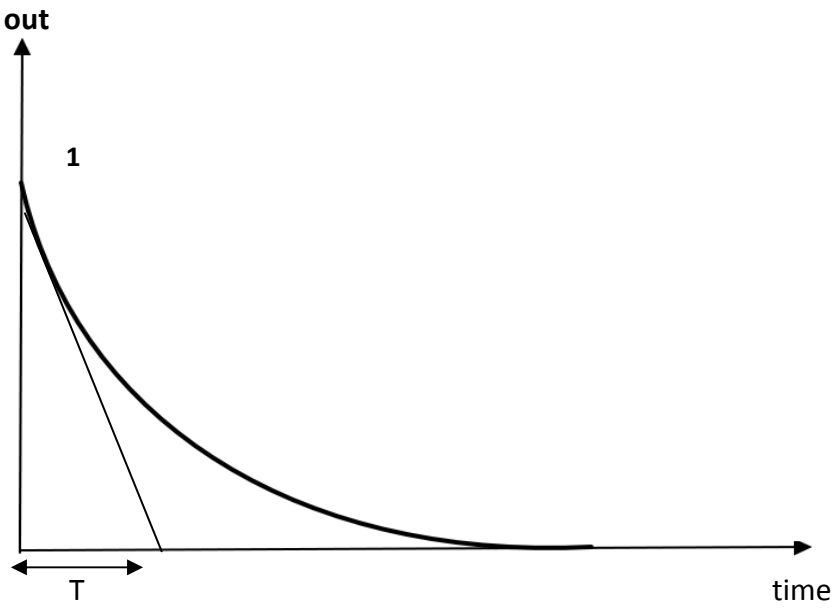
In simple words, higher frequency disturbances, noise, in signal are filtered.
The more **T** constant is bigger, the stronger works the filter.
T cannot be too big if we don't want to lose the dynamic
of the signal and worsening the control loop efficiency .

initialization
Not needed. Load start value into **in=out** in Arduino setup loop if required.
Default value is zero

Function Diff

call every 100ms using control function **Takt100**

parameters
float **in** input
float **out** output
float **T** [s]time constant



Response of **Diff** element to step **in** jump =1

Function **Diff** is a high pass filter, realizing the following Laplace function:

pT /1+pT

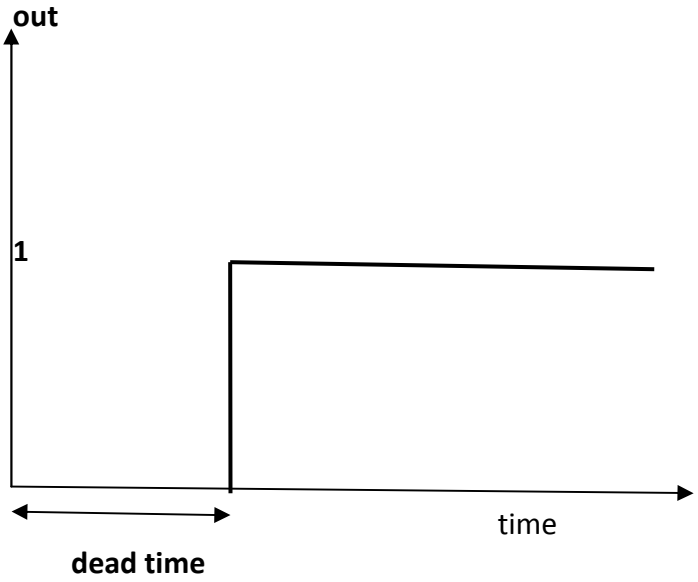
Using this filter we increase disturbances, noise, in signal.
Function **Diff** must be used carefully only for smooth signals if we want to increase their impact in control loop .
E.g. PID controllers deploy D term to strength the reaction of controllers for control difference variations.

initialization
No initialization.
Starting value of **out** is zero, if **in**=0.

Function Dead

Call every 1s using **Takt1000** - then the max dead time is 30s,
Using **Takt100** in 100 ms sampling we get 3 s max dead time.
By means of function we delay analog values on **output** in relation to **input** at a defined dead time.

parameters
float **in** input
float **out** output
int **deadtime** [s] dead time 0- 30 multiply sample time
float **inArray[31**] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
bool **start**

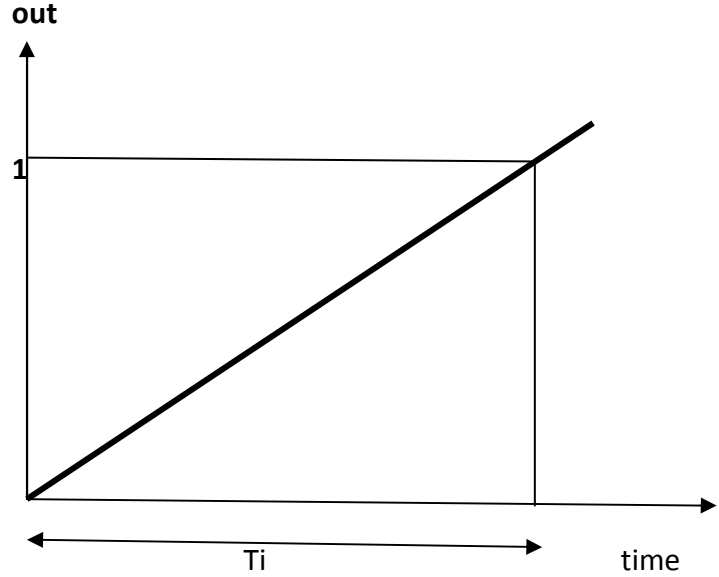


Response of **Dead** element to step **in** jump =1.

initialization
Bit **start=true** initializes **Dead** function and writes to **inArray** many equal input values "**in"** existing in moment of initialization. And these values appear first on output **Dead.** If this is not done, on output of **Dead** are appearing zeros during the time <= **deadtime.**

Function Int

Call every 100ms using **Takt100**.
parameters
in [0 -100] input
out [0 -100] output
Ti[s] integration time.



Response of **Int** element to step **in** jump =1, starting point zero: **out0**=0

Input **in** is integrated according to following formula:

out = ∫input.dt/Ti+ out0

Where **out(0**) is output value in time t=0 , which could be set in Arduino setup loop.
After setup every 100 ms new value is added:

delta **out = in***0.1/**Ti**

The input and output are limited in the range of 0 – 100.
In this library the integrator is used to simulate servomotor working on step controller output.

initialization
Not needed.
Change starting point loading value to **out** in Arduino setup loop if required.
Default value is 0.

Linearization Function Linear

This function does not need any time sampling.
parameters
float **LinArr[21]={0,5,10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85,90,95,100}**
float **in** input
float **out** output

Control loops work good without any nonlinearity, but especially they need a linear actual value.
In Arduino we process analog values in the following way.
On the input of Arduino we put a voltage in the range of 0 – 5V. We read this value by means of **analogRead** command, getting digital value 0 -1023.
Arduino controllers require 0 – 100.0 floating value, so the operation
act=**analog Read**/10.23 satisfies this demand.
Now remains the question, how the physical measured value is represented by our floating value 0 -100 ? We need a linear function between 0 -100 value and a physical value for indication and control loop – we should know how big must be the controller setpoint , to satisfy the target equation:

diff=set – act =0

There are some nonlinear sensors and transducers,
e.g. resistor PT100 or others sensors for the temperature measurement.
In such cases using of class **Linear** is a remedy.
In the array **LinArr** should be stored 21 interpolation points which correspond to input in the range from 0 – 100 in the following way:

0 **physical value limit down**
5 **physical value** interpolation point 1
10 **physical value** interpolation point 2

……

100 **physical value limit up**

Default points in program represent a linear characteristic.
Replace them through points of your nonlinear characteristic.

On the output of **Linear** we will get a linear physical value appropriate for indication in the range (**physical value limit down – physical value limit up**).
For the purpose of controllers we convert the result into the range of **0 -100** through the following operation:

act= out*100/(limit up – limit down)

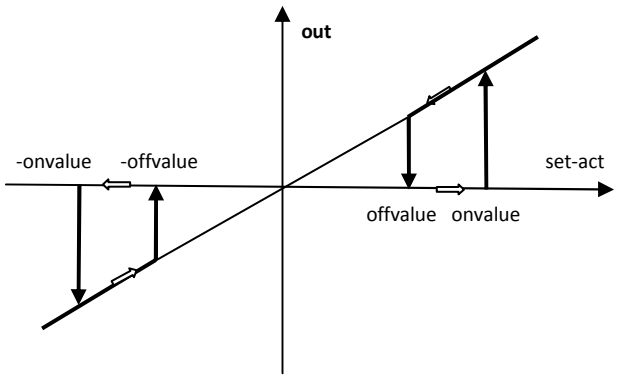
and define controller setpoint also in the range of 0 – 100.

initialization
Not needed.

Function Hyst

This function does not need any time sampling.

parameters
float **act** actual value
float **set** setpoint
float **out** output
float **onvalue** enable output = setpoint – actual value
float **offvalue** disable output = 0
Hyst creates hysteresis between actual value and setpoint according to the picture:



This function is useful for formation, near the equilibrium setpoint=actual value, dead zone and hysteresis in order to limit the action of controller and stabilize the control loop.
This hysteresis is built in controller **SPID**. No controller pulses are sent to servo motor if actual value almost reaches setpoint. This is defined through

parameters **onvalue** and **offvalue** (picture).

initialization
Not needed.

Function SetGen

Call every 100 ms using function **Takt100**.

parameters
float **set** [0 – 100]
bool **butt_up**
bool **butt_down**

float [%] **set_slow**
float [%] **set_fast**
float [s] **switch_ time**

This control function generates setpoint in a form of ramping.
If one of the two push buttons is pressed, the setpoint goes from the last **set** value (put it in Arduino setup loop) up or down.
In the beginning the ramping speed is **set_slow** in % and after the **switch_time** is over, the speed changes to **set_fast %**.
The setpoint is limited in the range of 0 – 100.

initialization
In Arduino setup loop this function is initialized automatically
Change starting point loading value to **set** in Arduino setup loop if required.
Default value is 0.

Function SetJpRamp

Call every 100 ms using function **Takt100.**

parameters
float **act** actual value
float **set** setpoint
float **out** output
float **speed_higher** [%]
float **speed_lower** [%]

bool **start** enable ramp

This is another function for smoothing jumps of setpoint **set**.
After new setpoint jump on input **set**, **SetJpRamp** generates on output **out**

ramps with different speed, depending on the direction of setpoint change.
Either ramp up, if new setpoint is bigger the last setpoint, with
speed **set_higher** %/s , or ramp down with speed **set_lower** %/s.
The ramping begins from actual value on input **act** and stops,
when the target setpoint is reached.
When the setpoint was changed once again during the ramping,
the reaction depends on the progress in ramping.
If the new setpoint is bigger as the the ramp value on **out** and the ramping
was direction higher, the ramp will continue moving.
If the new setpoint is lower **out**, a new ramp with a speed **set_lower**
direction down, is started.
By ramping down and setpoint change the action is similar, but into the
opposite direction.

initialization
Function is enabled if start =1. When **start** = 0, the function is switched off and
runs in bypass - then **out = set**.

Function SetTimeTable

Call every second using function **Time1000**.

parameters
float **set** setpoint
int **TimeMin[20]** = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
int **TimeSec**[20] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 };
float **SetArray**[20] = {50,50,50,50,50,50,50,50,50,50,50,50,50,50,50,50,50,50,50,50};

bool **start** starting the sequence

This function produces setpoint **set** due to 20 values of setpoint
stored in **SetArray a**nd time stored in arrays **TimeSec** and **TimeMin**.
After bit **start** is set, the program begins from the first value of setpoint in **SetArray**[0]
und puts it to the output **set** as long as the sum of time stored in both arrays **TimeMin**
and **TimeSec** in the same position is not over. Then switches to the next setpoint
from **SetArray**.

Example:
The first stored value in **SetArray**[0] is setpoint = 30, the second element **SetArray**[1] is
20, **TimeSec**[0]**=**0 (default) and **TimeMin**[0]=5 (see **examples**, simulations)
On the output of this function appears **set**=30 during first five minutes.
Then follows the value 20.

Plausibility of values in arrays.

There is no check of setpoint value, but the content of the two time arrays is checked in the following way:

1. If the sum of **TimeSec** and **TimeMin** values in the same position in arrays is zero, the sequence is stopped.
   Use this to terminate the time schedule. Write zero into both arrays to stop the sequence.

2. If the new sum **TimeSec** and **TimeMin** is smaller than the last value time, the sequence is stopped (error back in time movement attempt).

If the sequence is stopped, the value of setpoint set can be overwritten and changed.

initialization
Every time bit **start** =0 resets the sequence.
After bit **start** =1 sequence is running from the beginning according to the time table.