# 1. Introduction

This document describes the derivation of a PID controller that can be implemented in the brew application. The PID controller should be capable of controlling the temperature of the Hot Liquid Tun (HLT, 90 L) to within 0.5 °C.
The HLT contains a heating element of 3 kW, which is driven by the PID controller output signal Gamma [0..10 %]. The HLT temperature sensor is a LM92 12 bit + sign bit, ± 0.33°C accurate temperature sensor.

This document contains the following information

- Chapter 2: Derivation of a time-discrete algorithm for a PID controller
- Chapter 3: Derivation of an improved algorithm (a so-called 'type C' PID controller)
- Chapter 4: Description of algorithms for finding the optimum set of $K_c$, $T_i$, $T_d$ and $T_s$ values of the PID controller
- Chapter 5: Experimental results
- An appendix containing the C source listing

# 2. Derivation of a time-discrete algorithm for a PID controller

The generic equation[1] for a PID controller in the time-continuous domain is:

$$y(t) = K_c.\left( e(t) + \frac{1}{T_i}\int e(t)dt + T_d.\frac{de(t)}{dt} \right) \qquad \textit{eq. 01}$$

With:
$K_c = K_p$ Proportional Gain (for our temperature controller, unity is [% / °C])
$T_i = K_c / K_i$ Time-constant Integral gain [sec.]
$T_d = K_d / K_c$ Time-constant Derivative gain [sec.]
$T_s$ Sample period (default value is 5 seconds)
y(t) PID output signal, also called Gamma, ranges from [0..100 %]
e(t) error signal = set-point SP – process variable PV = $T_{set\_hlt} - T_{hlt}$

The corresponding equation in the s-domain is then:

$$\frac{Y(s)}{E(s)} = K_c.\left( 1 + \frac{1}{T_i.s} + T_d.s \right) \qquad \textit{eq. 02}$$

This transfer function has no real practical use, since the gain is increased as the frequency increases. Practical PID controllers limit this high frequency gain, using a first order low-pass filter. This results in the following transfer function:

$$\frac{Y(s)}{E(s)} = K_c.\left( 1 + \frac{1}{T_i.s} + \frac{T_d.s}{\gamma.s+1} \right) = K_c.\left( 1 + \frac{1}{T_i.s} + \frac{T_d}{\gamma + s^{-1}} \right) \qquad \textit{eq. 03}$$

where $\gamma$ is a small time-constant and may be set as 10% of the value of the derivative term $T_d$.

[1] Also known as the "Allen Bradley Logix5550 Independent PID equation"

Equation eq. 03 needs to transferred to the Z domain to make it suitable for implementation on a computer. This is done using the bilinear transformation (given in eq. 04):

The bilinear transformation formula is given with:

$$s = \frac{2}{T_s} \cdot \frac{1 - z^{-1}}{1 + z^{-1}} \qquad \textit{eq. 04}$$

Now use the bilinear transformation, given in equation *eq. 04,* to transform equation *eq. 03* onto an equivalent form in the Z –domain:

$$\frac{Y(z)}{E(z)} = K_c \cdot \left( 1 + \frac{T_s}{2.T_i} \cdot \frac{1 + z^{-1}}{1 - z^{-1}} + \frac{T_d \cdot \left(1 - z^{-1}\right)}{\gamma \cdot \left(1 - z^{-1}\right) + \frac{T_s}{2} \cdot \left(1 + z^{-1}\right)} \right) \qquad \textit{eq.05-1}$$

This transforms into:

$$\frac{Y(z)}{E(z)} = K_c \cdot \left( 1 + \frac{T_s}{2.T_i} \cdot \frac{1 + z^{-1}}{1 - z^{-1}} + 2.T_d \cdot \frac{1 - z^{-1}}{T_s + 2.\gamma + z^{-1} \cdot \left(T_s - 2.\gamma\right)} \right) \qquad \textit{eq.05-2}$$

This transforms into:

$$\frac{Y(z)}{E(z)} = K_c \cdot \left( 1 + \frac{T_s}{2.T_i} \cdot \frac{1 + z^{-1}}{1 - z^{-1}} + \frac{2.T_d}{T_s + 2.\gamma} \cdot \frac{1 - z^{-1}}{1 + z^{-1} \cdot \frac{T_s - 2.\gamma}{T_s + 2.\gamma}} \right) \qquad \textit{eq.05-3}$$

This transforms into:

$$\frac{Y(z)}{E(z)} \cdot \left(1 + (\alpha - 1) z^{-1} - \alpha.z^{-2}\right) = K_c \cdot \left( 1 + (\alpha - 1) z^{-1} - \alpha.z^{-2} + \frac{T_s}{2.T_i} \cdot \left(1 + (\alpha + 1) z^{-1} + \alpha.z^{-2}\right) + \frac{2.T_d}{T_s + 2.\gamma} \cdot \left(1 - 2.z^{-1} + z^{-2}\right) \right)$$

$$\text{with } \alpha = \frac{T_s - 2.\gamma}{T_s + 2.\gamma} \qquad \textit{eq.05-4}$$

Equation eq.05-4 is simplified into:

$$Y(z).(1 + (\alpha - 1).z^{-1} - \alpha.z^{-2}) = E(z).\left(k_0 + k_1.z^{-1} + k_2.z^{-2}\right) \qquad \textit{eq. 06}$$

With:

$$k_0 = K_c \cdot \left( 1 + \frac{T_s}{2.T_i} + \frac{2.T_d}{T_s + 2.\gamma} \right); k_1 = K_c \cdot \left( \alpha - 1 + \frac{T_s \cdot (\alpha + 1)}{2.T_i} - \frac{4.T_d}{T_s + 2.\gamma} \right); k_2 = K_c \cdot \left( \frac{\alpha.T_s}{2.T_i} - \alpha + \frac{2.T_d}{T_s + 2.\gamma} \right) \qquad \textit{eq. 07}$$

Transforming equation *eq. 06* back to the time-discrete form results in:

$$y_k = (1 - \alpha).y_{k-1} + \alpha.z^{-2} + k_0.e_k + k_1.e_{k-1} + k_2.e_{k-2} \qquad \textit{eq. 08}$$

Equation eq.08 is implemented with pid_reg2() and eq.07 is implemented with init_pid2() (see appendix for source listing).

# 3. Derivation of a Type C PID controller

There are three types of PID equations (see http://bestune.50megs.com/typeabc.htm) , with type C being the preferred one. Equation eq. 01 (and ultimately eq. 08) are type A equations, since the P- and the D-term both contain the set-point. Any changes in the set-point may cause an unwanted change in the PID output y(t).
Removing the set-point from the D-term results in a type B controller. The type C controller has also removed the set-point from the P-term, resulting in an even better PID controller implementation.

Starting with equation 01 and differentiating both sides gives equation eq. 09

$$dy(t) = K_c.\left(de(t) + \frac{e(t)}{T_i} + T_d.\frac{d^2e(t)}{dt}\right) \qquad \textit{eq. 09}$$

Transforming equation eq. 09 to the time-discrete domain, using backwards differentiation, results in equation eq. 10:

$$y_k = y_{k-1} + K_c.\left[(e_k - e_{k-1}) + \frac{T_s.e_k}{T_i} + \frac{T_d}{T_s}.(e_k - 2.e_{k-1} + e_{k-2})\right] \qquad \textit{eq. 10}$$

The D-term needs to be filtered with a Low-Pass Filter (LPF) to make it more practical. The transfer function of a simple LPF is given with:

$$H(s) = \frac{1}{\gamma.s + 1}$$ , with $\gamma$ typically set to about 10% of the $T_d$ value. *eq.11a*

The equivalent Z transfer function is:

$$H(z) = \frac{T_s.(1 + z^{-1})}{2.\gamma.(1 - z^{-1}) + T_s.(1 + z^{-1})} = \frac{T_s}{T_s + 2.\gamma}.\frac{1 + z^{-1}}{1 + \frac{T_s - 2.\gamma}{T_s + 2.\gamma}.z^{-1}} \qquad \textit{eq.11b}$$

The equivalent function is the time-discrete domain is then:

$$lpf_k = \frac{2.\gamma - T_s}{2.\gamma + T_s}.lpf_{k-1} + \frac{T_s}{T_s + 2.\gamma}.(e_k + e_{k-1}) \qquad \textit{eq.11c}$$

Equation eq.10 can now also be written as:

$$y_k = y_{k-1} + K_c.\left[(e_k - e_{k-1}) + \frac{T_s.e_k}{T_i} + \frac{T_d}{T_s}.(lpf_k - 2.lpf_{k-1} + lpf_{k-2})\right] \qquad \textit{eq. 12}$$

Equation 12 is still a type A equation ("textbook PID"), because the $K_c$ term depends on $e_k$ and the input of the LPF also has $e_k$ as input. Equation eq.11c and eq.12 are implemented with pid_reg3() and with init_pid3() (see appendix for source listing).

We will now transform equation eq.12 into a type C equation (eq. 13):

$$y_k = y_{k-1} + K_c.\left[-\left(PV_k - PV_{k-1}\right) + \frac{T_s.e_k}{T_i} - \frac{T_d}{T_s}.\left(lpf_k - 2.lpf_{k-1} + lpf_{k-2}\right)\right] \qquad eq.\ 13$$

Here, PV is the process variable, which is $T_{hlt}$ (the actual temperature of the HLT).

Furthermore, the input of the LPF should now be $T_{hlt}$ (and not $e_k$ as in equation eq.11c). Equations 11c and eq.13 are implemented with pid_reg4() and with init_pid4() (see appendix for source listing).

# 4. Finding the optimum set of PID parameters

Finding the optimum parameters for a PID controller can be difficult. Optimum means that the set-point temperature is reached as quickly as possible with overshoot minimised.

Three well-known algorithms for determining the PID parameters are described here:

- Ziegler-Nichols open-loop: set PID controller to a certain output and determine slope and dead-time of HLT system
- Ziegler-Nichols closed-loop: measure step-response
- Cohen-Coon: also a closed-loop method. Measure step-response
- Integral of the time weighted absolute error (ITAE): results in the best performance. The error signal is minimised (over time).

Some terms are frequently used in this document:

- **Dead-time Θ:** this is the time-delay between the initial step and a 10% increase in the process variable (the HLT temperature in our case).
- **$K_{hlt}$:** the gain of the HLT-system. The HLT-system receives the Gamma value (PID output) as input and has the HLT temperature as output. Unity of $K_{hlt}$ is [°C / %].
- **$\tau_{hlt}$:** the time-constant of the HLT-system. The HLT-system can be described with a first-order process model with time-delay (FOPTD). The transfer function for this model is:

$$HLT(s) = \frac{K_{hlt}.e^{-\theta.s}}{\tau_{hlt}.s+1}$$

- **a***: the normalised slope of the step response. Equal to ΔT / (Δt. Δp) with:
    - ΔT: change in temperature [°C]
    - Δt : change in time [seconds]
    - Δp: change in PID controller output [%]

With these three parameters, the optimum PID parameters are determined using table 1 on the next page (values are given both for PID operation and for PI-only operation):

| Method: / Parameter: | **$K_c$ [% / °C]** | **$T_i$ [seconds]** | **$T_d$ [seconds]** |
|---|---|---|---|
| **Ziegler-Nichols Open-loop** | $K_c = \frac{1{,}2}{\theta . a^*}$ | $T_i = 2{,}0.\theta$ | $T_d = 0{,}5.\theta$ |
| **Ziegler-Nichols Open-loop** | $K_c = \frac{0{,}9}{\theta . a^*}$ | $T_i = 3{,}33.\theta$ | **- -** |
| **Ziegler-Nichols Closed-loop** | $K_c = \frac{1{,}2.\tau_{hlt}}{K_{hlt}.\theta}$ | $T_i = 2{,}0.\theta$ | $T_d = 0{,}5.\theta$ |
| **Ziegler-Nichols Closed-loop** | $K_c = \frac{0{,}9.\tau_{hlt}}{K_{hlt}.\theta}$ | $T_i = 3{,}33.\theta$ | **- -** |
| **Cohen-Coon** | $K_c = \frac{\tau_{hlt}}{K_{hlt}.\theta} . \left( \frac{\theta}{4.\tau_{hlt}} + \frac{4}{3} \right)$ | $T_i = \theta . \frac{32.\tau_{hlt} + 6.\theta}{13.\tau_{hlt} + 8.\theta}$ | $T_d = \theta . \frac{4.\tau_{hlt}}{2.\theta + 11.\tau_{hlt}}$ |
| **Cohen-Coon** | $K_c = \frac{\tau_{hlt}}{K_{hlt}.\theta} . \left( \frac{\theta}{12.\tau_{hlt}} + \frac{9}{10} \right)$ | $T_i = \theta . \frac{30.\tau_{hlt} + 3.\theta}{9.\tau_{hlt} + 20.\theta}$ | - - |
| **ITAE-Load** | $K_c = \frac{1{,}357}{K_{hlt}} . \left( \frac{\theta}{\tau_{hlt}} \right)^{-0{,}947}$ | $T_i = \frac{\tau_{hlt}}{0{,}842} . \left( \frac{\theta}{\tau_{hlt}} \right)^{0{,}738}$ | $T_d = 0{,}381.\tau_{hlt} . \left( \frac{\theta}{\tau_{hlt}} \right)^{0{,}995}$ |
| **ITAE-Load** | $K_c = \frac{0{,}859}{K_{hlt}} . \left( \frac{\theta}{\tau_{hlt}} \right)^{-0{,}977}$ | $T_i = \frac{\tau_{hlt}}{0{,}674} . \left( \frac{\theta}{\tau_{hlt}} \right)^{0{,}680}$ | - - |

Table 1: optimum PID parameters for the various methods

To be able to find these three parameters accurately, two experiments need to be conducted. These two experiments are described in the next two paragraphs.
The last paragraph (§ 4.3) shows the calculated PID parameters for all these methods.

## 4.1 Experiment 1: Determine dead-time Θ of HLT-system

Manually set the PID-controller output ("gamma") to a certain value (e.g. 20 %). In case of a heavy load, 100 % is recommended (more accurate), but if you don't know the performance of the system, a lower value to start with is better. The temperature starts to increase and follows the curve in figure 1.

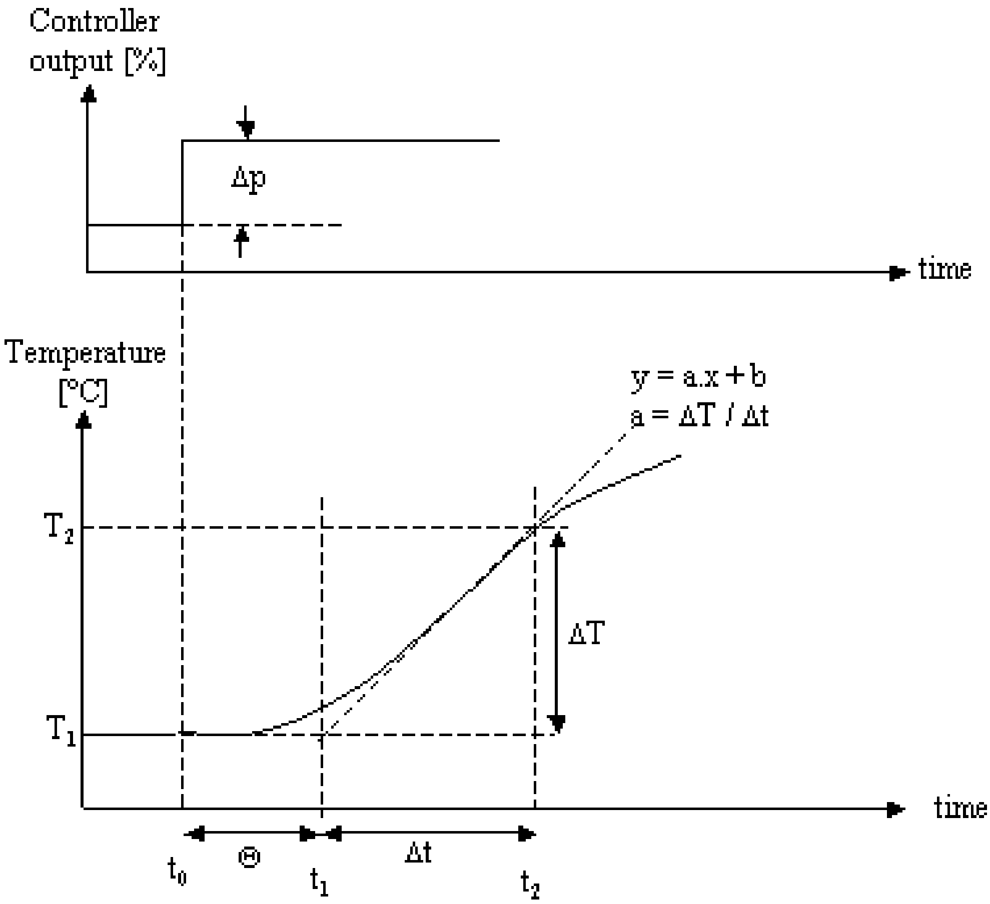


Figure 1: Open-loop response of the system

Calculate the average slope of the rise ($a = \Delta T / \Delta t$) by using regression analysis if possible.
Calculate the normalised slope a*, which is defined as: $a^* = a / \Delta p$.
The dead-time Θ is defined as the time between $t_0$ and $t_1$ (by using regression analysis, this can be done quite accurately).

Experimental data:[2] HLT filled with 85 L water, lid off

| | |
|---|---|
| 19:08:20 | $t_0$, the PID controller output gamma was set to 100 %, $T_1$ = 47,41 °C |
| | Step-response up to 55.00 °C, $\Delta T$ = 7,59 °C |
| 19:12:25 | $T_{hlt}$ = 48,18 °C ($\geq T_1$ + 10%.$\Delta T$ = 48,17 °C) |
| 19:27:21 | $T_{hlt}$ = 54,25 °C ($\geq T_1$ + 90%.$\Delta T$ = 54,24 °C) |

Regression analysis of the data between 19 :12 :25 and 19:27 :21 resulted in the following:

$y = a.x + b = 0{,}0334x + 48{,}278$ $\qquad R^2 = 0{,}9995$

Here, every data-point for x represents 5 seconds. Therefore, the average slope a is equal to:

a = 0,0334 °C / 5 sec. = 6,68E-03 °C/second (which is 0,4 °C/minute)

Now solve where this curve hits the Tenv line:

[2] Measurements are recorded in HLT_open_loop_response_260404.xls

x = (47.41 – 48.278) / 6.68E-03 = -130 seconds (or 2 minutes and 10 seconds)
The dead-time moment is then 19:12:25 – 2:10 = 19:10:15
Therefore, the **dead-time Θ = 19:10:15 – 19:08:20 = 1:55 = 115 seconds**

The normalised slope a* is equal to :

$a^* = a / \Delta p = 6{,}68E\text{-}05$ °C/(%.second)

## 4.2 Experiment 2: Determine gain $K_{hlt}$ and time-constant $\tau_{hlt}$ of HLT-system

Manually set the PID-controller output ("gamma") to a certain value (e.g. 20 %). The temperature starts to increase and follows the curve in figure 2.

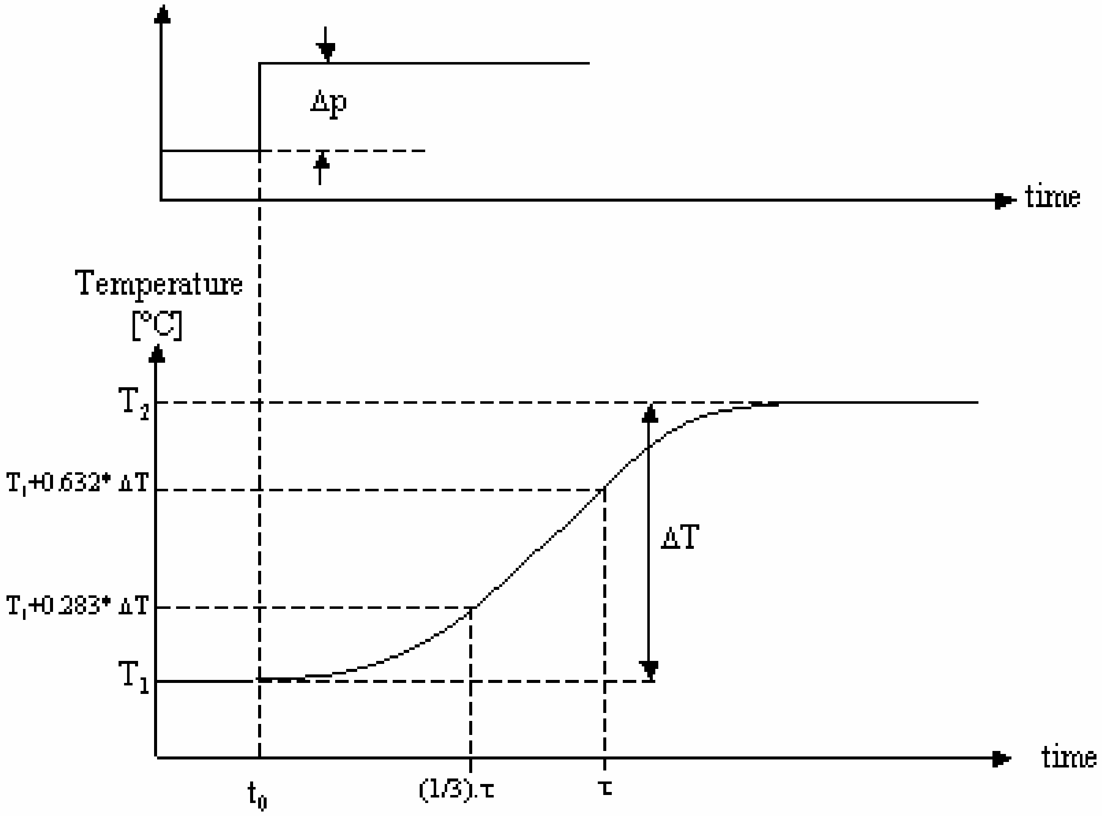


Figure 2: Step response of the system

Because of the large time-constant presumably present in the HLT system, the step response is not very accurate in determining the dead-time **Θ**. This is the mean reason to conduct two experiments (theoretically, the step-response would give you all the required information to determine the three parameters).

Experimental data:[3] HLT filled with 85 L water, lid off

| | |
|---|---|
| 09:49:04 | $t_0$, the PID controller output gamma was set to 20 %, $T_1$ = 19,20 °C |
| 20:24:03 | Experiment stopped, T = 52,98 °C. |
| | Regression analysis (2nd order polynomial) used to find maximum. |
| | Maximum found at 21:20:44 (8300 ticks of 5 seconds). **$T_2$ = 52,989 °C** |
| | $\Delta T$ = 52,989 – 19,20 = 33,789 °C |
| | $T(t = 1/3.\tau_{hlt}) = T_1 + 0{,}283 * \Delta T$ = 28,762 °C => $1/3.\tau_{hlt}$ = 6599 seconds |
| | $T(t = \tau_{hlt}) = T_1 + 0{,}632 * \Delta T$ = 40,555 °C => $\tau_{hlt}$ = 16573 seconds |
| | Solve for $\tau_{hlt}$: $\tau_{hlt} - 1/3.\ \tau_{hlt}$ = 16573 – 6599 => **$\tau_{hlt}$ = 14961 seconds** |

Solve for Gain $K_{hlt}$: $K_{hlt} = \Delta T / \Delta p$ = 33,789 °C / 20 % = **1,689 °C / %**

[3] Measurements are recorded in HLT_step_response_290404.xls

## 4.3 Calculation of PID parameters for the various methods

In the previous paragraphs, the following parameters were found:

- **dead-time Θ** **= 115 seconds**
- **Gain $K_{hlt}$** **= 1,689 °C / %**
- **Time-constant $\tau_{hlt}$** **= 14961 seconds**
- **Normalised slope a*** **= a / Δp = 6,68E-05 °C/(%.second)**

| | Kc [%/°C] | Ti [sec] | Td [sec] | |
|---|---|---|---|---|
| Ziegler-Nichols Open Loop | 156,2 | 230,0 | 57,5 | PID |
| | 117,2 | 383,0 | - - | PI |
| Ziegler-Nichols Closed Loop | 92,4 | 230,0 | 57,5 | PID |
| | 69,3 | 383,0 | - - | PI |
| Cohen-Coon | 102,8 | 282,2 | 41,8 | PID |
| | 69,4 | 377,2 | - - | PI |
| ITAE-Load | 80,8 | 489,0 | 44,9 | PID |
| | 59,2 | 810,2 | - - | PI |

Table 2: Calculation of parameters for the various methods

## Version History

| Date | Version | Description |
|---|---|---|
| 01-02-2003 | V1.0 | First version for display on web-site |
| 08-03-2004 | V2.0 | - Removed implementation 1 and 3<br>- Added description of types A, B and C<br>- Added type C PID algorithm<br>- Added auto-tuning algorithm + example |
| 10-03-2004 | V2.1 | Incorrect dead-time calculation |
| 09-05-2004 | V2.2 | - Added Cohen-Coon + ITAE methods<br>- Added two experiments + description<br>- Calculations are updated |
| 10-05-2004 | V3.0 | - Derivation with Taylor series replaced by exact derivation. Current implementation worked well, but simulation showed problems |
| 13-05-2004 | V3.01 | - Update of document, corrected a few mistakes<br>- Add low-pass filter to the D-term of the PID-controllers |
| 13-05-2004 | V3.02 | Some textual changes |
| 09-05-2006 | V3.03 | Foutje verbeterd in equation eq.09 |

## Appendix I: C-programs (pid_reg.h, pid_reg.c)

```
/*==================================================================
  File name    : $Id: pid_reg.h,v 1.7 2004/05/13 20:51:00 emile Exp $
  Author       : E. vd Logt
  ------------------------------------------------------------------
  Purpose : This file contains the defines for the PID controller.
  ------------------------------------------------------------------
  $Log: pid_reg.h,v $
  Revision 1.7  2004/05/13 20:51:00  emile
  - Main loop timing improved. Only 99 (of 100) cycles were executed. Loop
    timing is now reset after 100 loops (5 seconds)
  - TS parameter now only works on PID-controller time-slice. Max. is 20 sec.
  - Bug-fix in init_ma() filter when init. to a value (should be /N).
  - LPF for D-term of PID controller added. New reg. var. K_LPF
  - PID Debug label added + checkbox in PID screen. Default off (NO reg var).
  - Statusbar object added
  - Start made with network functionality. Not operational yet.

  Revision 1.6  2004/05/10 20:54:30  emile
  - Bug-fix: log-file header: '\n' was removed, is corrected now
  - Hints added to PID dialog screen
  - Now four PID controllers to choose from. Old ebrew version is still listed,
    but should not be used anymore. Simulation showed stability problems.
    Preferably use the type C controller.

  Revision 1.5  2004/05/08 14:52:52  emile
  - Mash pre-heat functionality added to STD. New registry variable PREHEAT_TIME.
    tset_hlt is set to next mash temp. if mash timer >= time - PREHEAT_TIME
  - View mash progress screen: reorganised, pre-heat timers added, timers are now
    in seconds instead of minutes.
  - update_tset() function removed. Now incorporated in STD, states 3-5 + (new state) 13.
  - THLT_HLIMIT and THLT_LLIMIT and state 4 'Bypass Heat Exchanger' removed
  - Reorganisation of several variables (e.g. ms_idx, ms_tot) into (other) structs.
  - 'Apply' Button added to Fix parameters dialog screen.
  - 'Edit mash scheme' no longer resets the (running) mash timers
  - 'Mash progress controlled by' function removed. Registry var 'mash_control' now
    also removed.
  - Changing init. volume of HLT did not result in an update on screen. Corrected.

  Revision 1.4  2004/05/05 15:44:16  emile
  - Main Screen picture update
  - Init_ma() now initialises with a value instead of 0. Avoids reset of signal.
  - STD update: calculation of volumes should be correct now
  - Parameter added for early start of mash timer. Registry var. TOffset2
  - Registry variables Kc, Ti, Td, TOffset and TS are now floats instead of integers.
  - Some screens updated with hints (also of labels)
  - Bug-fix: unnecessary delay after change in gamma. Is corrected now
  - Help via menu now also works

  Revision 1.3  2004/04/25 14:02:17  emile
  - Added a 'type C' PID controller, function pid_reg3(). Possible to select
    this from the PID settings dialog screen. Left the old one in for
    compatibility. New registry variable PID_Model.
  - Gamma added to log-file, so that the PID controller can be tuned.

  Revision 1.2  2002/12/30 13:33:45  emile
  - Headers with CVS tags added to every source file
  - Restore Settings function is added
  - "ebrew" registry key now in a define REGKEY

  V0.1 060302 First version
  ==================================================================
*/
#ifndef PID_REG_H
#define PID_REG_H

#ifdef __cplusplus
extern "C" {
#endif

// These defines are needed for loop timing and PID controller timing
#define TWENTY_SECONDS (400)
#define TEN_SECONDS    (200)
#define FIVE_SECONDS   (100)
#define ONE_SECOND      (20)
#define T_50MSEC  (50)   // Period time of TTimer in msec.

#define GMA_HLIM (100.0) // PID controller upper limit [%]
#define GMA_LLIM   (0.0) // PID controller lower limit [%]

typedef struct _pid_params
{
   double kc; // Controller gain from Dialog Box
```

```
   double ti; // Time-constant for I action from Dialog Box
   double td; // Time-constant for D action from Dialog Box
   double ts; // Sample time [sec.] from Dialog Box
   double k_lpf; // Time constant [sec.] for LPF filter
   double k0; // k0 value for PID controller
   double k1; // k1 value for PID controller
   double k2; // k2 value for PID controller
   double k3; // k3 value for PID controller
   double lpf1; // value for LPF filter
   double lpf2; // value for LPF filter
   int    ts_ticks;  // ticks for timer
   int    pid_model; // PID Controller type [0..3]
   double pp; // debug
   double pi; // debug
   double pd; // debug
} pid_params; // struct pid_params

//--------------------
// Function Prototypes
//--------------------
void init_pid1(pid_params *p);
void pid_reg1(double xk, double *yk, double tset, pid_params *p, int vrg);
void init_pid2(pid_params *p);
void pid_reg2(double xk, double *yk, double tset, pid_params *p, int vrg);
void init_pid3(pid_params *p);
void pid_reg3(double xk, double *yk, double tset, pid_params *p, int vrg);
void init_pid4(pid_params *p);
void pid_reg4(double xk, double *yk, double tset, pid_params *p, int vrg);

#ifdef __cplusplus
};
#endif
#endif
```

```
/*==================================================================
  Function name: init_pid1(), pid_reg1(), init_pid2(), pid_reg2()
                 init_pid3(), pid_reg3(), init_pid4(), pid_reg4()
  Author       : E. vd Logt
  File name    : $Id: pid_reg.c,v 1.6 2004/05/13 20:51:00 emile Exp $
  ------------------------------------------------------------------
  Purpose : This file contains the main body of the PID controller.
            For design details, please read the Word document
            "PID Controller Calculus".

            In the GUI, the following parameters can be changed:
            Kc: The controller gain
            Ti: Time-constant for the Integral Gain
            Td: Time-constant for the Derivative Gain
            Ts: The sample period [seconds]
  ------------------------------------------------------------------
  $Log: pid_reg.c,v $
  Revision 1.6  2004/05/13 20:51:00  emile
  - Main loop timing improved. Only 99 (of 100) cycles were executed. Loop
    timing is now reset after 100 loops (5 seconds)
  - TS parameter now only works on PID-controller time-slice. Max. is 20 sec.
  - Bug-fix in init_ma() filter when init. to a value (should be /N).
  - LPF for D-term of PID controller added. New reg. var. K_LPF
  - PID Debug label added + checkbox in PID screen. Default off (NO reg var).
  - Statusbar object added
  - Start made with network functionality. Not operational yet.

  Revision 1.5  2004/05/10 20:54:30  emile
  - Bug-fix: log-file header: '\n' was removed, is corrected now
  - Hints added to PID dialog screen
  - Now four PID controllers to choose from. Old ebrew version is still listed,
    but should not be used anymore. Simulation showed stability problems.
    Preferably use the type C controller.

  Revision 1.4  2004/04/26 13:30:22  emile
  Bug-fix: init_pid3() did not calculate correctly when Ti = 0. Corrected.

  Revision 1.3  2004/04/25 14:02:17  emile
  - Added a 'type C' PID controller, function pid_reg3(). Possible to select
    this from the PID settings dialog screen. Left the old one in for
    compatibility. New registry variable PID_Model.
  - Gamma added to log-file, so that the PID controller can be tuned.

  Revision 1.2  2002/12/30 13:33:45  emile
  - Headers with CVS tags added to every source file
  - Restore Settings function is added
  - "ebrew" registry key now in a define REGKEY

  ==================================================================
*/
#include "pid_reg.h"

static double ek_1;  // e[k-1]  = SP[k-1] - PV[k-1] = Tset_hlt[k-1] - Thlt[k-1]
static double ek_2;  // e[k-2]  = SP[k-2] - PV[k-2] = Tset_hlt[k-2] - Thlt[k-2]
static double xk_1;  // PV[k-1] = Thlt[k-1]
static double xk_2;  // PV[k-2] = Thlt[k-1]
static double yk_1;  // y[k-1]  = Gamma[k-1]
static double yk_2;  // y[k-2]  = Gamma[k-1]
static double lpf_1; // lpf[k-1] = LPF output[k-1]
static double lpf_2; // lpf[k-2] = LPF output[k-2]

void init_pid1(pid_params *p)
/*------------------------------------------------------------------
  Purpose  : This function initialises the PID controller, based on
             implementation method 2 (Bilinear Transformation with Taylor series)
             Do not use this version!
  Variables: p: pointer to struct containing all PID parameters
                         Ts      Td
             k0 = Kc.(1 + ---- + 2.--)
                        2.Ti     Ts

                          Ts    6.Td
             k1 = Kc.(-1 + ---- - ----)
                          2.Ti    Ts

                  8.Kc.Td
             k2 = --------
                     Ts
  Returns  : No values are returned
  ------------------------------------------------------------------*/
{
   p->ts_ticks = (int)((p->ts * 1000.0) / T_50MSEC);
   if (p->ts_ticks > TWENTY_SECONDS)
   {
      p->ts_ticks = TWENTY_SECONDS;
   }
   if (p->ti == 0.0)
```

```
   {
      p->k0 = p->kc * (+1.0 + (2.0 * p->td / p->ts));
      p->k1 = p->kc * (-1.0 - (6.0 * p->td / p->ts));
   }
   else
   {
      p->k0 = p->kc * (+1.0 + (p->ts / (2.0 * p->ti)) + (2.0 * p->td / p->ts));
      p->k1 = p->kc * (-1.0 + (p->ts / (2.0 * p->ti)) - (6.0 * p->td / p->ts));
   } // else
   p->k2 = p->kc * 8.0 * p->td / p->ts;
} // init_pid1()

void pid_reg1(double xk, double *yk, double tset, pid_params *p, int vrg)
/*------------------------------------------------------------------
  Purpose  : This function implements the PID controller, conform
             implementation method 2 ("Bilinear Transformation with Taylor Series").
             Do not use this version!
             This function should be called once every TS seconds.
  Variables:
        xk : The input variable x[k] (= measured temperature)
       *yk : The output variable y[k] (= gamma value for power electronics)
      tset : The setpoint value for the temperature
        *p : Pointer to struct containing PID parameters
        vrg: Release signal: 1 = Start control, 0 = disable PID controller
  Returns  : No values are returned
  ------------------------------------------------------------------*/
{
   double ek; // e[k]

   ek = tset - xk;  // calculate e[k]
   if (vrg)
   {
      *yk += (p->k0 * ek);   // y[k] = y[k-1] + k0 * e[k]
      *yk += (p->k1 * ek_1); //               + k1 * e[k-1]
      *yk += (p->k2 * ek_2); //               + k2 * e[k-2]
   }
   else *yk = 0.0;

   ek_2  = ek_1; // e[k-2] = e[k-1]
   ek_1  = ek;   // e[k-1] = e[k]

   // limit y[k] to GMA_HLIM and GMA_LLIM
   if (*yk > GMA_HLIM)
   {
      *yk = GMA_HLIM;
   }
   else if (*yk < GMA_LLIM)
   {
      *yk = GMA_LLIM;
   } // else
} // pid_reg1()

void init_pid2(pid_params *p)
/*------------------------------------------------------------------
  Purpose  : This function initialises the PID controller, based on
             the new Type A PID controller. Update of pid_reg1().
  Variables: p: pointer to struct containing all PID parameters
                           Ts        2.Td
             k0 = Kc.(1 + ---- + ------------)
                          2.Ti   Ts + 2.k_lpf

                                 Ts.(alfa + 1)       4.Td
             k1 = Kc.(alfa - 1 + -------------- - ------------)
                                      2.Ti       Ts + 2.k_lpf

                             alfa.Ts       2.Td
             k2 = Kc.(-alfa + ------- + ------------)
                               2.Ti    Ts + 2.k_lpf

                         Ts - 2.k_lpf
             with alfa = ------------
                         Ts + 2.k_lpf

  Returns  : No values are returned
  ------------------------------------------------------------------*/
{
   double alfa = (p->ts - 2.0 * p->k_lpf) / (p->ts + 2.0 * p->k_lpf); // help variable

   p->ts_ticks = (int)((p->ts * 1000.0) / T_50MSEC);
   if (p->ts_ticks > TWENTY_SECONDS)
   {
      p->ts_ticks = TWENTY_SECONDS;
   }
   if (p->ti == 0.0)
   {
      p->k0 = p->kc * (+1.0 + (2.0 * p->td / (p->ts + 2.0 * p->k_lpf)));
      p->k1 = p->kc * (alfa - 1.0 - (4.0 * p->td / (p->ts + 2.0 * p->k_lpf)));
```

```
      p->k2 = p->kc * (-alfa + (2.0 * p->td / (p->ts + 2.0 * p->k_lpf)));
   }
   else
   {
      p->k0 = p->kc * (+1.0 + (p->ts / (2.0 * p->ti))
                            + (2.0 * p->td / (p->ts + 2.0 * p->k_lpf))
                      );
      p->k1 = p->kc * (alfa - 1.0 +
                       (p->ts * (alfa + 1.0) / (2.0 * p->ti)) -
                       4.0 * p->td / (p->ts + 2.0 * p->k_lpf)
                      );
      p->k2 = p->kc * (-alfa + (alfa * p->ts / (2.0 * p->ti))
                                 + (2.0 * p->td / (p->ts + 2.0 * p->k_lpf))
                      );
   } // else
   //-----------------------------------------------
   // y[k] = (1- alfa)*y[k-1] + alfa*y[k-2] + ....
   //-----------------------------------------------
   p->lpf1 = 1.0 - alfa;
   p->lpf2 = alfa;
} // init_pid2()

void pid_reg2(double xk, double *yk, double tset, pid_params *p, int vrg)
/*------------------------------------------------------------------
  Purpose  : This function implements the updated PID controller.
             It is an update of pid_reg1(), derived with Bilinear
             Transformation. It is a Type A controller.
             This function should be called once every TS seconds.
  Variables:
        xk : The input variable x[k] (= measured temperature)
       *yk : The output variable y[k] (= gamma value for power electronics)
      tset : The setpoint value for the temperature
        *p : Pointer to struct containing PID parameters
        vrg: Release signal: 1 = Start control, 0 = disable PID controller
  Returns  : No values are returned
  ------------------------------------------------------------------*/
{
   double ek; // e[k]

   ek = tset - xk;  // calculate e[k]
   if (vrg)
   {
      *yk  = p->lpf1 * yk_1 + p->lpf2 * yk_2; // y[k] = (1-alfa)*y[k-1] + alfa*y[k-2]
      *yk += p->k0 * ek;                      //  ... + k0 * e[k]
      *yk += p->k1 * ek_1;                    //  ... + k1 * e[k-1]
      *yk += p->k2 * ek_2;                    //  ... + k2 * e[k-2]
   }
   else *yk = 0.0;

   ek_2 = ek_1; // e[k-2] = e[k-1]
   ek_1 = ek;   // e[k-1] = e[k]

   // limit y[k] to GMA_HLIM and GMA_LLIM
   if (*yk > GMA_HLIM)
   {
      *yk = GMA_HLIM;
   }
   else if (*yk < GMA_LLIM)
   {
      *yk = GMA_LLIM;
   } // else

   yk_2 = yk_1; // y[k-2] = y[k-1]
   yk_1 = *yk;  // y[k-1] = y[k]
} // pid_reg2()

void init_pid3(pid_params *p)
/*------------------------------------------------------------------
  Purpose  : This function initialises the Allen Bradley Type A PID
             controller.
  Variables: p: pointer to struct containing all PID parameters

                   Kc.Ts
             k0 =  -----   (for I-term)
                    Ti

                       Td
             k1 = Kc . --  (for D-term)
                       Ts

             The LPF parameters are also initialised here:
             lpf[k] = lpf1 * lpf[k-1] + lpf2 * lpf[k-2]
  Returns  : No values are returned
  ------------------------------------------------------------------*/
{
   p->ts_ticks = (int)((p->ts * 1000.0) / T_50MSEC);
   if (p->ts_ticks > TWENTY_SECONDS)
```

```
   {
      p->ts_ticks = TWENTY_SECONDS;
   }
   if (p->ti == 0.0)
   {
      p->k0 = 0.0;
   }
   else
   {
      p->k0 = p->kc * p->ts / p->ti;
   } // else
   p->k1   = p->kc * p->td / p->ts;
   p->lpf1 = (2.0 * p->k_lpf - p->ts) / (2.0 * p->k_lpf + p->ts);
   p->lpf2 = p->ts / (2.0 * p->k_lpf + p->ts);
} // init_pid3()

void pid_reg3(double xk, double *yk, double tset, pid_params *p, int vrg)
/*------------------------------------------------------------------
  Purpose  : This function implements the type Allen Bradley Type A PID
             controller. All terms are dependent on the error signal e[k].
             The D term is also low-pass filtered.
             This function should be called once every TS seconds.
  Variables:
        xk : The input variable x[k] (= measured temperature)
       *yk : The output variable y[k] (= gamma value for power electronics)
      tset : The setpoint value for the temperature
        *p : Pointer to struct containing PID parameters
        vrg: Release signal: 1 = Start control, 0 = disable PID controller
  Returns  : No values are returned
  ------------------------------------------------------------------*/
{
   double ek;  // e[k]
   double lpf; //LPF output

   ek = tset - xk;  // calculate e[k] = SP[k] - PV[k]
   //--------------------------------------
   // Calculate Lowpass Filter for D-term
   //--------------------------------------
   lpf = p->lpf1 * lpf_1 + p->lpf2 * (ek + ek_1);

   if (vrg)
   {
      //-----------------------------------------------------------
      // Calculate PID controller:
      // y[k] = y[k-1] + Kc*(e[k] - e[k-1] +
      //                     Ts*e[k]/Ti +
      //                     Td/Ts*(lpf[k] - 2*lpf[k-1]+lpf[k-2]))
      //-----------------------------------------------------------
      p->pp = p->kc * (ek - ek_1);  // y[k] = y[k-1] + Kc*(e[k] - e[k-1])
      p->pi = p->k0 * ek;           //      + Kc*Ts/Ti * e[k]
      p->pd = p->k1 * (lpf - 2.0 * lpf_1 + lpf_2);
      *yk += p->pp + p->pi + p->pd;
   }
   else *yk = 0.0;

   ek_1  = ek;    // e[k-1] = e[k]
   lpf_2 = lpf_1; // update stores for LPF
   lpf_1 = lpf;

   // limit y[k] to GMA_HLIM and GMA_LLIM
   if (*yk > GMA_HLIM)
   {
      *yk = GMA_HLIM;
   }
   else if (*yk < GMA_LLIM)
   {
      *yk = GMA_LLIM;
   } // else
} // pid_reg3()

void init_pid4(pid_params *p)
/*------------------------------------------------------------------
  Purpose  : This function initialises the Allen Bradley Type C PID
             controller.
  Variables: p: pointer to struct containing all PID parameters
  Returns  : No values are returned
  ------------------------------------------------------------------*/
{
   init_pid3(p); // identical to init_pid3()
} // init_pid4()

void pid_reg4(double xk, double *yk, double tset, pid_params *p, int vrg)
/*------------------------------------------------------------------
  Purpose  : This function initialises the Allen Bradley Type C PID
             controller, the P and D term are no longer dependent on
             the set-point, only on PV (which is Thlt).
             The D term is also low-pass filtered.
```

```
             This function should be called once every TS seconds.
  Variables:
        xk : The input variable x[k] (= measured temperature)
       *yk : The output variable y[k] (= gamma value for power electronics)
      tset : The setpoint value for the temperature
        *p : Pointer to struct containing PID parameters
        vrg: Release signal: 1 = Start control, 0 = disable PID controller
  Returns  : No values are returned
  ------------------------------------------------------------------*/
{
   double ek;  // e[k]
   double lpf; //LPF output

   ek = tset - xk;  // calculate e[k] = SP[k] - PV[k]
   //--------------------------------------------------------------
   // Calculate Lowpass Filter for D-term: use x[k] instead of e[k]!
   //--------------------------------------------------------------
   lpf = p->lpf1 * lpf_1 + p->lpf2 * (xk + xk_1);

   if (vrg)
   {
      //-----------------------------------------------------------
      // Calculate PID controller:
      // y[k] = y[k-1] - Kc*(PV[k] - PV[k-1] +
      //                      Ts*e[k]/Ti -
      //                      Td/Ts*(lpf[k] - 2*lpf[k-1]+lpf[k-2]))
      //-----------------------------------------------------------
      p->pp = -p->kc * (xk - xk_1);  // y[k] = y[k-1] - Kc*(PVk - PVk-1)
      p->pi = p->k0 * ek;            //      + Kc*Ts/Ti * e[k]
      p->pd = -p->k1 * (lpf - 2.0 * lpf_1 + lpf_2);
      *yk += p->pp + p->pi + p->pd;
   }
   else { *yk = p->pp = p->pi = p->pd = 0.0; }

   xk_1  = xk;    // PV[k-1] = PV[k]
   lpf_2 = lpf_1; // update stores for LPF
   lpf_1 = lpf;

   // limit y[k] to GMA_HLIM and GMA_LLIM
   if (*yk > GMA_HLIM)
   {
      *yk = GMA_HLIM;
   }
   else if (*yk < GMA_LLIM)
   {
      *yk = GMA_LLIM;
   } // else
} // pid_reg4()
```