21. PID control

The PID (Proportional Integral Differential) controller is a basic building block in regulation. It can be implemented in many different ways, this example will show you how to code it in a microcontroller and give a simple demonstration of its abilities.

21.1. The theory of PID control

Consider a well stirred pot of water (system), which we need to keep at the desired temperature (reference value, *R*) above the temperature of the surroundings. What we do is we insert a thermometer (sensor) into water and read its temperature (actual value, *X*). If the water is too cold, we turn-on the heater (actuator) placed under the pot. Once the temperature reading on the thermometer reaches the desired value we turn off the heater. The temperature of the water still rises for some time (overshoot), and then starts to decrease. When temperature of the water drops below the desired value we turn-on the heater again. It takes some time before the heater heats-up (this causes an undershoot in temperature) and starts to deliver the heat into the water, but eventually the temperature of the water reaches the desired value again, and the process repeats. What we have is a regulation system, where we act as a controller; we observe the actual value, compare it with the reference value, and stimulate the system based on the result of the comparison, Fig. 21.1.

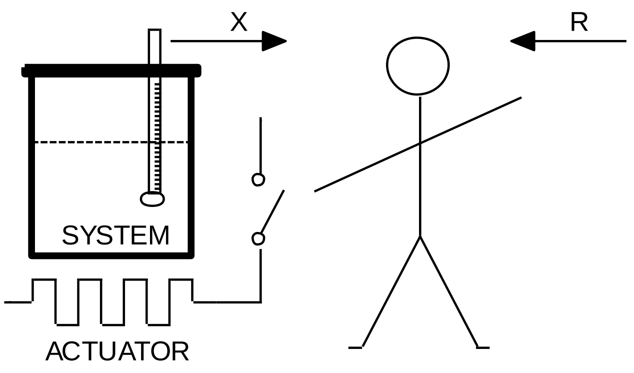

Figure 21.1: A crude example for a regulation

The temperature of the water in the above example never remains at the desired value, but instead wobbles around it. The wobbling depends on the properties of the system, and properties of the sensor and actuator. In order to improve the behavior of the temperature and reduce the wobbling we can improve the regulation process by introducing more complex decisions in the controller. For instance: we could stop the heating some time before the temperature reaches the desired value if we know the amount of overshoot. We could reduce the overshoot also by reducing the amount of heat delivered into the water when the actual temperature becomes close to the desired. There are other possibilities, but they can all be put to life by introduction of a control unit which performs so-called PID regulation.

In terms of regulation theory the above crude system including the actuator and sensor can be described by a second order differential equation, and the regulated system is called a second order. These are best tamed by a PID controller, Figure 22.2.

A PID controller consists first of a unit to calculate the difference (Error) between the desired value Ref and the actual value X. The calculated error signal is fed to three units to calculate the multiple of the error (proportional part, Prop), the rate of changing of the error (differential part, Dif), and the up-to-now sum of the error (integral part, Int). All three components are weighted by corresponding factors (Kp, Kd, Ki) and summed to get the final value (Reg) used by the actuator to influence the system.

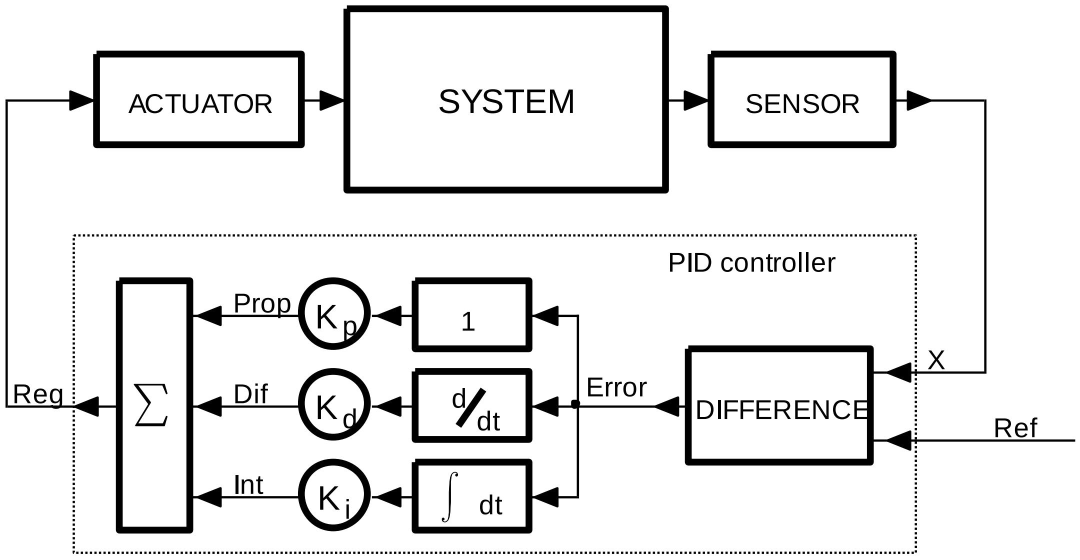

Figure 22.2: A system controlled by a PID controller

21.2. The PID controller implemented in a microprocessor

The microcontroller houses all required building blocks to implement the measurement of the actual and reference value (ADC), the computation of the controller function, and the generation of a signal to influence the system (DAC). When such PID controller is implemented in microcontroller the calculation cannot be continuous as with analog electronic circuits, but must be performed periodically with a period that is short enough compared to the response time of the regulated system. This again calls for periodic sampling, calculation and generation of values. The same programming skeleton as used in FIR and IIR filtering can be re-used for the task of PID control. The initialization of the microcontroller is the same, and all calculation of the controller functions should be performed within the interrupt function. Its listing is given below, the calculation follows Figure 22.2 exactly.

```
void ADC_IRQHandler(void)              // this takes approx 6us of CPU time!
{
  GPIOE->ODR |=  0x0100;               // PE08 up
  R[k]  = ADC2->DR;                    // pass ADC -> circular buffer Ref
  X[k]  = ADC1->DR;                    // pass ADC -> circular buffer X

  Error[k] = R[k] - X[k];              // calculate error
  Prop = Kp * (float)Error[k];         // proportional part
  Dif  = Kd * (float)(Error[k] - Error[(k-1) & 63]) / Ts;  // differential part
  Int += Ki * (float)Error[k];         // integral part
  Reg[k] = (int)(Prop + Dif + Int);    // summ all three
  if (Reg[k] > 4095) DAC->DHR12R1  = 4095;    // limit output due to the DAC              // 13
  else if (Reg[k] < 0) DAC->DHR12R1  = 0;                                                 // 14
       else DAC->DHR12R1  = (int)Reg[k];      // regulator output -> DAC                  // 15
  DAC->DHR12R2  = Error[k] + 2048;            // Error -> DAC                             // 16
  k = (k + 1) & 63;                     // increment pointer to circular buffer           // 17
  GPIOE->ODR &= ~0x0100;                // PE08 down                                      // 18
}
```

The reference and the actual signals are measured by two ADC, and are stored in corresponding circular buffers *R* and *X* in lines 4 and 5. The calculation of the PID control is performed in lines 7 to 11, and results are sent to DACs in lines 13 to 16. The calculation is complemented by line 17 to increase and bound the pointer to circular buffer size, and by lines 3 and 18 to toggle port E, pin 8. This signal can be used to determine the execution time of the interrupt function.

Lines 7 to 11 are discussed next. First the error signal is calculated in line 7, being the difference between the desired and the actual value. The error signal is stored in its own circular buffer for further use. Next three components *Prop*, *Dif*, and *Int* are calculated. The component *Prop* is just a weighted error value, the weight being *Kp*. The component *Dif* should be a differential of the error value. Here we calculate the differential as a simple difference between the current error and the past error, divided by the time interval and weighted by value *Kd*. The last component is called *Int*, the integral of the error value. This is calculated by simple adding of the weighted current error to the sum of all previous errors; the weight is called *Ki*. Line 11 sums all three components together to form the controller output *Reg*.

Weights *Kp*, *Kd* and *Ki* are set in the main program, and are floating point values. The samples from ADCs are short integer values, and must be converted to floats before the use. The components *Prop*, *Dif* and *Int* are floating point values, and should be declared as such. The sum of all three components is again a floating point value, but should be sent to DAC as a short integer.

The result of PID controller may be outside the range of the DAC; it is best to limit the output from the controller to a value between 4095 and 0, as shown in lines 13 and 14. If the value is already within the range then we can simply send it there as in line 15. The error value is sent to the second DAC for evaluation in line 16; since it may be of either polarity it is best to offset it at the middle of the DAC range.

The listing of the rest of the program is presented next.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"
#include "dd.h"
#include "LCD2x16.c"

#define pi 3.141592653589

int   Ref[64], X[64], k;                // declare circular buffers                       // 12
float Kp = 1.0, Ki = 0.0, Kd = 0.0;     // declare & init params                          // 13
int Error[64];                          // declare error vector                           // 14
int Reg[64];                            // declare past output vector                     // 15
float Prop, Dif, Int = 0;               // declare (& init) vars                          // 16
float Ts = 0.0001;                      // defined by constant 8400 in TIM2->arr;         // 17
```

```
int main ()  {                                                                          // 19
int sw;
  GPIOEinit ();                                                                         // 21
  SWITCHinit();
  ADCinit_T5_CC1_IRQ();
  DACinit();
  TIM5init_TimeBase_CC1(8400);           // 8400 == 100us == 10kHz                      // 25

  LCD_init();                                                                           // 27
  LCD_string("Kp:", 0x00);                                                              // 28
  LCD_string("Kd:", 0x09);                                                              // 29
  LCD_string("Ki:", 0x49);                                                              // 30

  while (1) {                                                                           // 32
    sw = GPIOE->IDR;                                                                    // 33
    if ((sw & S370) && !(sw & S372) && !(sw & S373)) Kp--;   // manually set Kp         // 34
    if ((sw & S371) && !(sw & S372) && !(sw & S373)) Kp++;                              // 35
    if (Kp<0) Kp = 0;     if (Kp > 1000) Kp = 1000;                                     // 36
    if ((sw & S370) &&  (sw & S372) && !(sw & S373)) Kd -= 0.001;  // manually set Kd   // 37
    if ((sw & S371) &&  (sw & S372) && !(sw & S373)) Kd += 0.001;                       // 38
    if (Kd < 0) Kd = 0;   if (Kd > 1) Kd = 1;                                           // 39
    if ((sw & S370) && !(sw & S372) &&  (sw & S373)) Ki -= 0.0001; // manually set Ki   // 40
    if ((sw & S371) && !(sw & S372) &&  (sw & S373)) Ki += 0.0001;                      // 41
    if (Ki < 0) Ki = 0;   if (Ki > 1) Ki = 1;                                           // 42
    LCD_sInt3DG((int)Kp,0x03,1);                          // write Kp                   // 43
    LCD_sInt3DG((int)(Kd*1000),0x0c,1);                   // write Kd                   // 44
    LCD_sInt3DG((int)(Ki*10000),0x4c,1);                  // write Ki                   // 45
    for (int i = 0; i < 5000000; i++)      {};            // waste time                 // 46
  };
}
```

The listing starts with the inclusion of CMSIS files and the declaration of circular buffers in line 12. The weights are declared and initialized next, followed by the declaration of other variables in lines 13 to 17. The line 17 declares and initializes the time interval between successive samples to 100 μs.

The function "main()" starts with calls to the configuration functions for ADC, DAC, timer TIM5 and ports. All functions used were already described in previous chapters. The timer TIM5 is initialized to issue interrupt requests every 100 μs in line 25.

Lines 27 to 30 configure the LCD screen and write introductory text to it.

The quality of PID regulation depends on the values of weights. These can be set by pressing pushbuttons on the BaseBoard. Due to the limited number of pushbuttons a combination of them must be pressed to set the desired value. All this is handled in lines 34 to 42. Pushbutton S370 is used to decrease, and pushbutton S371 to increase the value of the weight selected. The weight to be changed is selected by the combination of pushbuttons S372 and S373:

- If none of them is pressed we are dealing with weight *Kp*,
- If S372 is pressed and S373 is not pressed we are dealing with weight *Kd*,
- If S372 is not pressed and S373 is pressed we are dealing with weight *Ki*.

Line 34 for instance checks the three pushbuttons S370, S372 and S373, and decreases the value of weight *Kp* is correct situation is detected. Line 35 is used to check the state of pushbuttons for increasing of the same weight. Line 36 is used to limit the value of this weight to a reasonable value in case the user requests too much. All three weights are written to the LCD screen in lines 43 to 45.

The program can be tested by adding a simple second order system between the DAC output (Reg, DAC1) and ADC input (actual value, ADC1->DR, ADC_IN1 input). The other ADC (ADC2->DR, ADC_IN2 input) is used to read the reference value Ref. Two serially connected RC circuits are used as a substitute for the second order system; this simplifies the demonstration. Additionally, the desired value Ref can be generated using a function generator, as can the interference signal Intf. The complete connection of the demo circuit is given in Fig. 22.3.

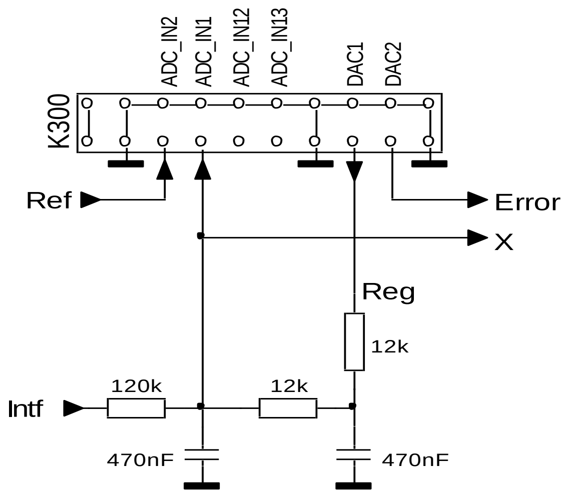

Figure 22.3: The connection for experiment

Figures 22.4 to 22.7 give the actual values (X, red) for the circuit in Fig. 22.3 for different values of K_p, K_d, and K_i. The interference signal Intf is kept at zero, and the desired value (Ref, blue) is a squarewave with the frequency of 10Hz. The offset of the Ref signal is set close to the middle of the ADC scale. The horizontal scale is in seconds, while the vertical scale is in volts.

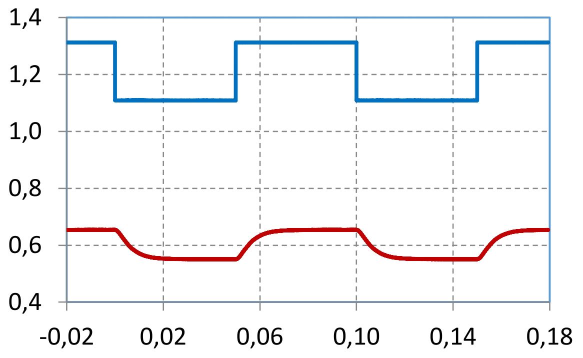
Figure 22.4: Kp = 1, Kd = 0, Ki = 0; Proportinal gain is too low and the actual value (red) does not reach the desired value (blue)

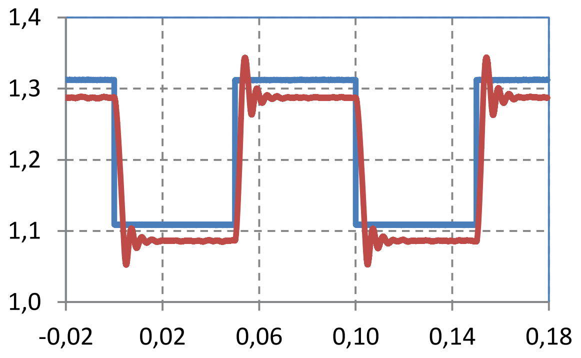
Figure 22.5: Kp = 50, Kd = 0, Ki = 0; Proportional gain is high, the actual value (red) is closer to the desired value (blue), but oscillations become visible

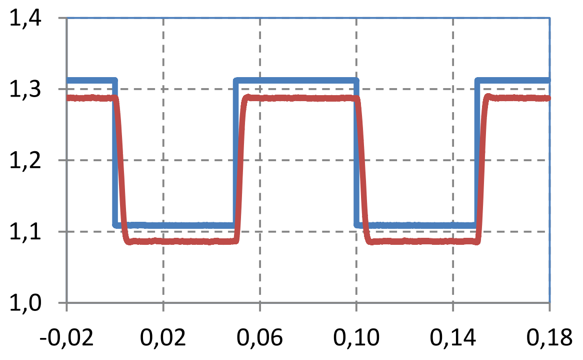
Figure 22.6: Kp = 50, Kd =40, Ki = 0; Differential gain smooth out oscillations, but the actual value (red) is still offseted from the desired value (blue)

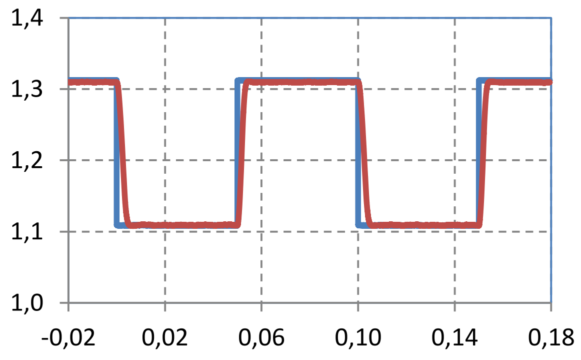
Figure 22.7: Kp = 50, Kd = 40, Ki = 40; The integral part pushes the actual value up to become almost identical to the desired value

The next set of diagrams on Figures 22.8 to 22.11 give responses (red) of the regulated system to the interference signal (blue) while the reference signal (not shown) is kept at a constant value of

1.21V. It is expected that the response is also constant if the regulator does its job properly. Scales are the same as for the previous set of figures.

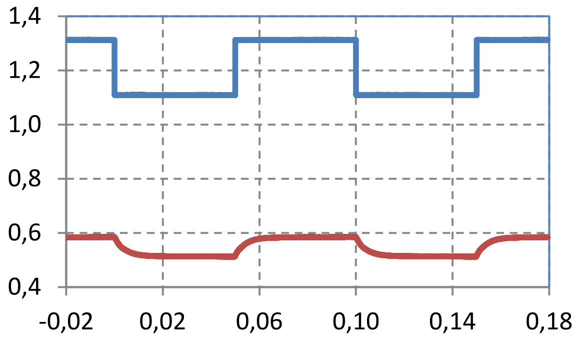

Figure 22.8: Kp = 1, Kd = 0, Ki = 0; Proportional gain is too low and the influence of the interfering signal is significant

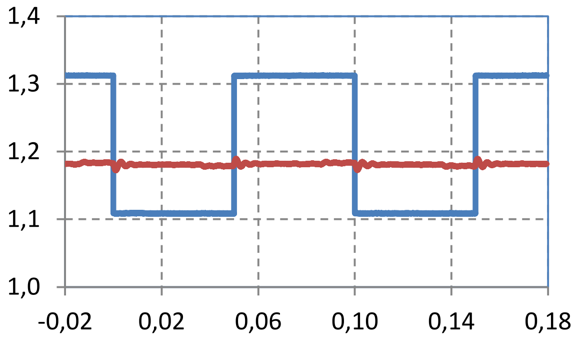

Figure 22.9: Kp = 50, Kd = 0, Ki = 0; Proportional gain is high, the actual value (red) is closer to 1.21V, but oscillations caused by the interfering signal are visible

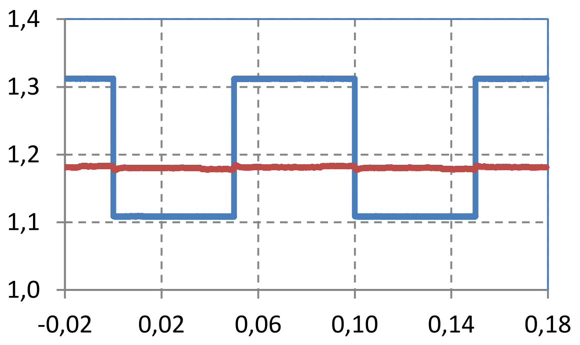

Figure 22.10: Kp = 50, Kd =4 0, Ki = 0; Differential gain smooth out the oscillations, but the actual value (red) is still not the same as the desired value (1.21V)

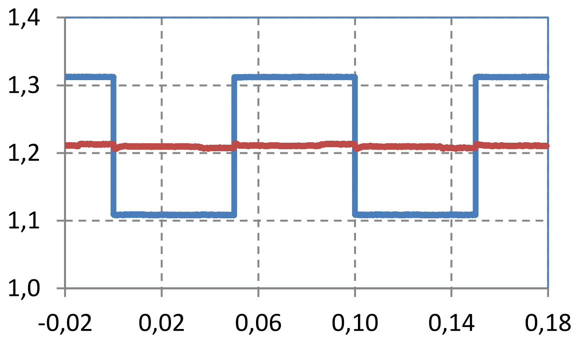

Figure 22.11: Kp = 50, Kd = 40, Ki = 40; The integral gain pushes the average of the actual value up to become the same as the average of the desired value (1.21V)