21. PID controller

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.

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 the water and read its temperature (actual value, X). If the water is too cold, then 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 the undershoot) 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 influence the system based on the result of the comparison, Fig. 1.
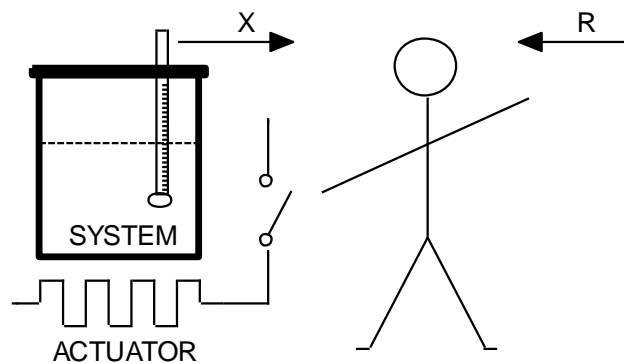

Figure 1: A crude example for a regulation

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 F 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 overshoots. 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 example 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.
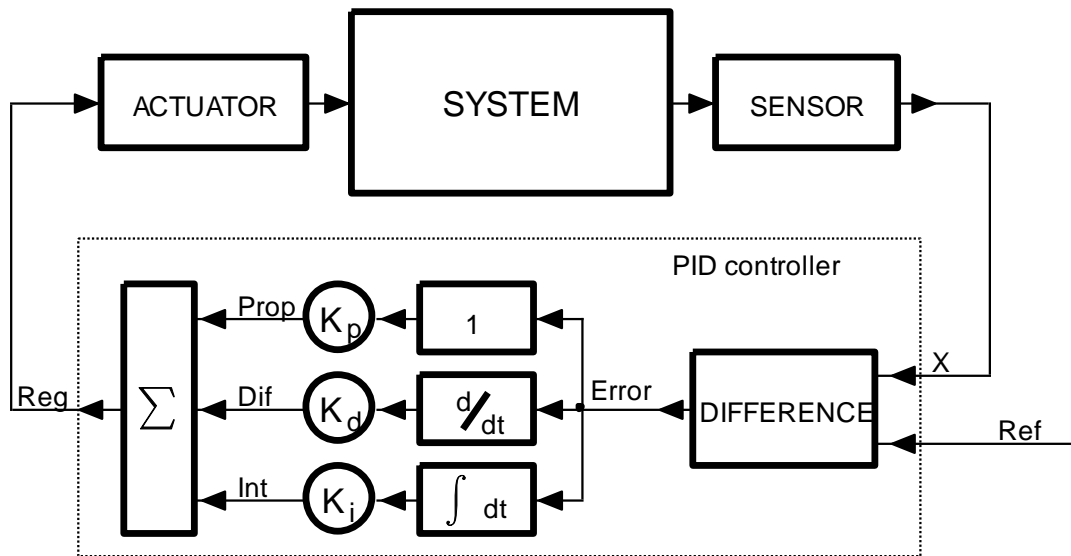

Figure 2: A system controlled by a PID controller

A PID controller consists first of a unit to calculate the difference between the desired value and the actual value. 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 (K_p, K_d, K_i) and summed to get the final value (Reg) used by the actuator to influence the system.

When such PID controller is implemented in microcontroller the above action must be performed periodically, the period being 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. The initialization of the microcontroller is the same, all calculation of the controller functions should be performed within the interrupt function. The listing of the program is given in Fig. 3.

The program starts with the declaration (and initialization where appropriate) of variables. Two integer circular buffers are used for the desired and actual value, and additional two floating point circular buffers for the error and past output values. Three variables for components and three for corresponding weights are declared as floating point numbers and initialized next. Finally, a variable needed to calculate the rate of change is declared and initialized.

Within the main function the ADC and DAC are initialized, the timer is programmed to issue one start of conversion at the ADCs every 100us giving the sampling rate of 10 kHz, and the interrupt controller NVIC is enabled for interrupt requests from the ADC. Following this the microcontroller continues with the execution of the endless loop. Since we are implementing a PID regulator it seems normal to allow the user to change the weight for proportional, differential and integral component during the execution of the program. The status of buttons is periodically checked, the period being defined by the time-wasting loop at the beginning of the endless loop. Next three lines of the program are used to change the value of the weight for the proportional component. If pushbutton S370 (connected to port E, bit 0, 0x01) is pressed in combination with the pushbutton S375

2

(connected to port E, bit 5, 0x20), then the weight is increased by one. If pushbutton S370 is pressed in combination with the pushbutton S374 (port E, bit 4, 0x10) then the weight is decreased by one. The third line bounds the value of the proportional weight to values between (and including) 0 and 1000. Next three lines do the same with the weight for differential component, and the next three lines for the integral component. Finally all three weights are written to the LCD screen.

```c
#include "stm32f4xx.h"
#include "LCD2x16.c"

int  Ref[64], x[64], Ptr;               // declare circular buffers
int Error[64], Reg[64];                 // declare error and past output vectors
float Prop, Dif, Int = 0;               // declare (& init) vars
float Kp = 1.0, Ki = 0.0, Kd = 0.0;     // declare & init params
float Ts = 0.0001;               // defined by constant 8400 in TIM2->arr; 10kHz

int main ()  {
  GPIO_setup();                  // GPIO set-up
  DAC_setup();                   // DAC set-up
  ADC_setup();                   // ADC set-up
  Timer2_setup();                // Timer 2 set-up
  NVIC_EnableIRQ(ADC_IRQn);      // Enable IRQ for ADC in NVIC

  LCD_init();
  LCD_string("Kp:", 0x00);   LCD_string("Kd:", 0x09);   LCD_string("Ki:", 0x49);

  // set gains & waste time - indefinite loop
  while (1) {
  for (int i = 0; i < 2000000; i++)      {};                  // waste time
  if ((GPIOE->IDR & 0x003f) == (0x01 + 0x20)) Kp++;       // manually set Kp
  if ((GPIOE->IDR & 0x003f) == (0x01 + 0x10)) Kp--;
  if (Kp<0) Kp = 0;           if (Kp > 1000) Kp = 1000;
  if ((GPIOE->IDR & 0x003f) == (0x02 + 0x20)) Kd += 0.001;  // manually set Kd
  if ((GPIOE->IDR & 0x003f) == (0x02 + 0x10)) Kd -= 0.001;
  if (Kd < 0) Kd = 0;         if (Kd > 1)     Kd = 1;
  if ((GPIOE->IDR & 0x003f) == (0x04 + 0x20)) Ki += 0.0001; // manually set Ki
  if ((GPIOE->IDR & 0x003f) == (0x04 + 0x10)) Ki -= 0.0001;
  if (Ki < 0) Ki = 0;         if (Ki > 1)     Ki = 1;
  LCD_sInt3DG((int)Kp,0x03,1);                              // write Kp
  LCD_sInt3DG((int)(Kd*1000),0x0c,1);                       // write Kd
  LCD_sInt3DG((int)(Ki*10000),0x4c,1);                      // write Ki
  };
}

// IRQ function
void ADC_IRQHandler(void)        // this takes approx 6us of CPU time!
{
  GPIOE->ODR |=  0x0100;                 // PE08 up
  Ref[Ptr] = ADC1->DR;                   // pass ADC -> circular buffer x1
  x[Ptr]   = ADC2->DR;                   // pass ADC -> circular buffer x2
  // PID calculation start
  Error[Ptr] = Ref[Ptr] - x[Ptr];                      // calculate error
  Prop = Kp * (float)Error[Ptr];                        // proportional part
  Dif  = Kd * (float)(Error[Ptr] - Error[(Ptr-1) & 63]) / Ts;  // diff. part
  Int += Ki * (float)Error[Ptr];                        // integral part
  Reg[Ptr] = (int)(Prop + Dif + Int);                   // summ all three
  // PID calculation stop
  if (Reg[Ptr] > 4095) DAC->DHR12R1  = 4095;    // limit output due to DAC
  else if (Reg[Ptr] < 0) DAC->DHR12R1  = 0;
       else DAC->DHR12R1  = (int)Reg[Ptr];      // regulator output -> DAC
  DAC->DHR12R2  = Error[Ptr] + 2048;            // Error -> DAC
  Ptr = (Ptr + 1) & 63;           // increment pointer to circular buffer
  GPIOE->ODR &= ~0x0100;          // PE08 down
}
```

Figure 3:The listing of a program to implement PID controller

The complete calculation of the control is performed within the interrupt function. The results of conversion from two ADCs (desired and actual value) are first stored in circular buffers. The difference between these two values is calculated next, and immediately saved into the third circular buffer as Error. Next three lines calculate the three components and weight them; the fourth line adds components together and saves the sum into the fourth circular buffer as Reg. This value should be the output of the regulator andi therefore sent to the DAC. However, the DAC can accept values between 0 and 4095, while other numbers are folded into this range; for instance the number 4097 is DAC-ed the same as number 1, and this would cause significant error in regulation. The numbers are therefore best limited to the acceptable values, and this is done in the next three lines of code. The second DAC is used to generate the analog version of the error signal, and the value is shifted for one half of the DAC range by adding 2048 to the error value. Finally, the pointer in circular buffers gets updated.

The calculation of the control is put between two statements to change values of bit 0, port E, and the signal generated at this bit can be used to determine the execution time of the interrupt function, which in this case is about 6μs.

The program can be tested by adding a simple second order system between the DAC output (Reg, DAC1) and ADC input (actual value, ADC2->DR, ADC3 input). The other ADC (ADC1->DR, ADC2 input) is used to read the desired value. Two serially connected RC circuits can be used as a substitute for the second order system; this simplifies the demonstration. Additionally, the

Figure 4: The connection for experiment

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. 4.
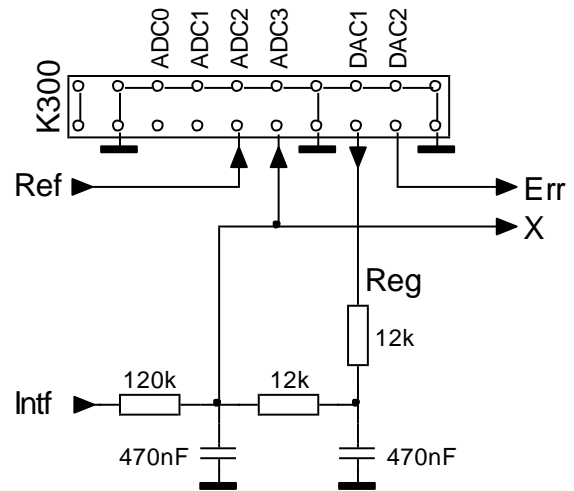
Figures 5 to 8 give the actual values (X, red) for the circuit in Fig. 4 for different values of K_p, K_d, and K_i. The interference signal Intf is kept at zero, and the desired value (Ref, blue) is a squarewave with the frequency of 10Hz. The offset of the Ref signal is set close to the middle of the ADC scale. The horizontal scale is in seconds, while the vertical scale is in volts.
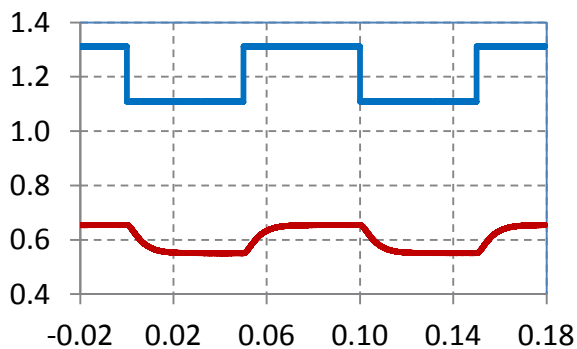

Figure 5: Kp = 1, Kd = 0, Ki = 0; Proportinal gain is too low and the actual value (red) does not reach the desired value (blue)
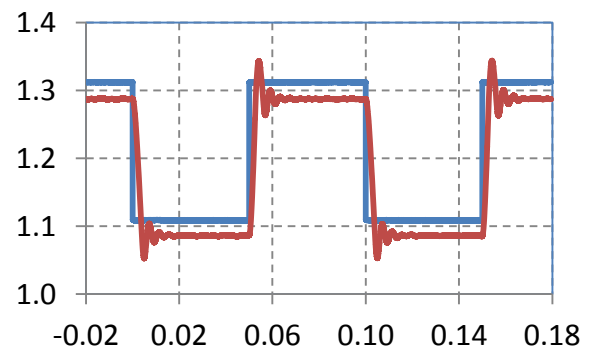

Figure 6: Kp = 50, Kd = 0, Ki = 0; Proportional gain is high, the actual value (red) is closer to the desired value (blue), but oscillations become visible

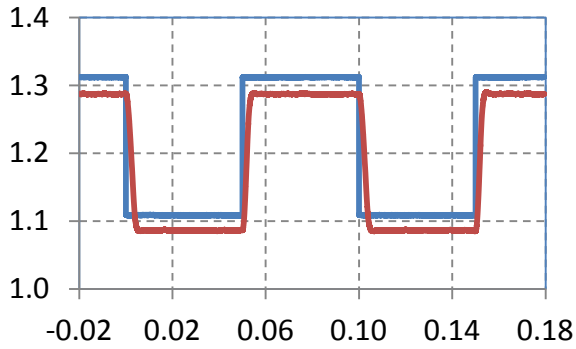*Figure 7: Kp = 50, Kd =4 0, Ki = 0; Differential gain smoothes out the oscillations, but the actual value (red) is still not the same as the desired value (blue)*
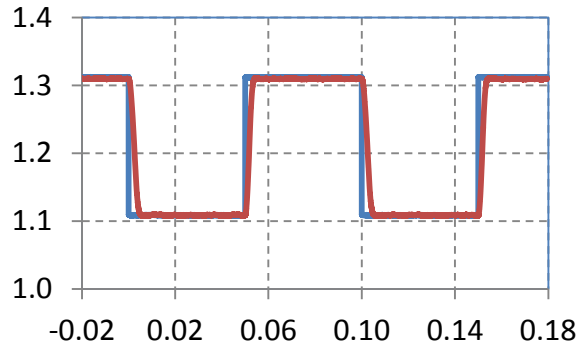

Figure 8: Kp = 50, Kd = 40, Ki = 40; The integral gain pushes the average of the actual value up to become the same as the average of the desired value

The next set of diagrams on Figures 9 to 13 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 a constant if the regulator does its job properly.
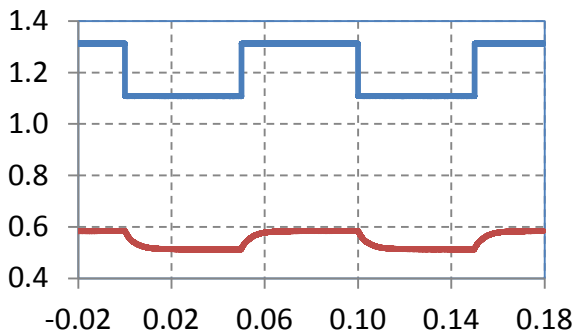

Figure 9: Kp = 1, Kd = 0, Ki = 0; Proportional gain is too low and the influence of the interfering signal is significant

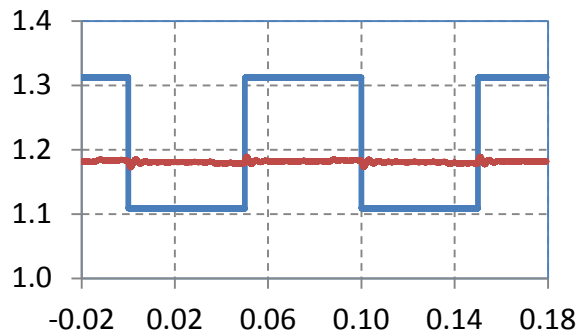*Figure 10: Kp = 50, Kd = 0, Ki = 0; Proportional gain is high, the actual value (red) is closer to 1.21V, but oscillations caused by the interfiring signal are visible*
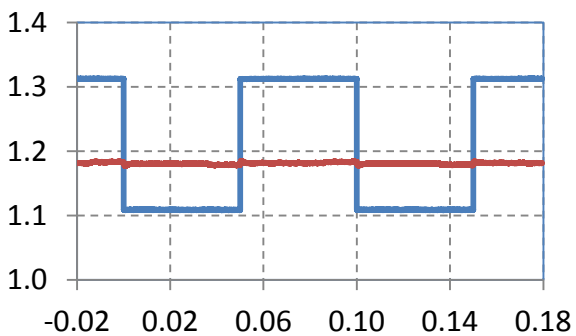

Figure 11: Kp = 50, Kd =4 0, Ki = 0; Differential gain smoothes out the oscillations, but the actual value (red) is still not the same as the desired value (1.21V)
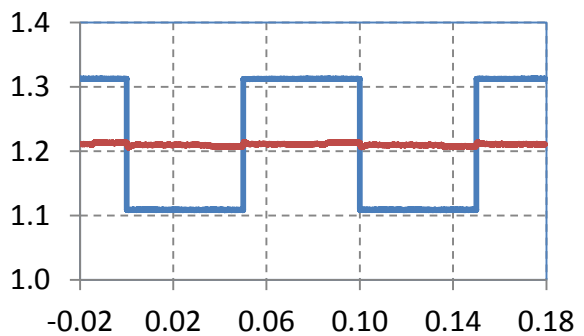

Figure 8: Kp = 50, Kd = 40, Ki = 40; The integral gain pushes the average of the actual value up to become the same as the average of the desired value (1.21V)