0.1 Softwaredesign of a PI controller

In order to implement the aforementioned PI-controller in a fashion that complies with API from FreeRTOS, it is chosen that a single interface is to be designed. This is due to the mattter that only one subprogram has to be written, thus it requieres to be generic, if multiple PI-controllers are to be implemented. To meet the given requirements the following design criteria must be fulfilled.

- Must comply with FreeRTOS's API
- Must be re-entrant
- Must be able to be initialized from main
- Must be generic

In order to write a routine that will act as the PI-controller, it must be wrapped into a never ending loop. In this the desired calculations will be done. This is due to meet the first criteria and is achieved in 0.1.

```
1  void tsk_PI(void* pvArgs);
2
3  void tsk_PI(void* pvArgs)
4  {
5      PI controller = *((PI*)pvArgs);          //use pvArgs to set up the PI
6      controller.conditional_anti = 1;
7      int16_t error = 0;
8
9      for (;;)
10     {
11         error = calculate_error_PI(controller.ref, controller.measured_y_n, &controller );
12         calculate_PI( error, &controller );
13
14         vTaskDelay(pdMS_TO_TICKS(100));
```

tsk_PI() will be the function which the task created by FreeRTOS will have as an argument. It is noticed that all of the above criteria are met, if the functions calculate_error_PI() and calculate_P() are assumed to be re-entrant. The third and fourth criteria are achieved by passing the struct PI from the function task_PI().

The struct will contain all the necessary gains and data-samples, which a PI-controller will need in order to operate. The struct members can be seen in 0.1.

```
1  }
2
3  typedef struct PI_dif_eq
4  {
5      int16_t x_n[3];
6      float y_n[3];
7      float kp;
8      float ki;
9      float sample_rate;
10     int16_t saturation;
11     int16_t* ref;
12     int16_t* PWM_out;
13     int16_t* measured_y_n;
14     int16_t* clamp_ref;
```

Since the design requirements are met, the next step is to implement the funtionallity of a PI-controller, which has the following requirements:

- Must be implemented as a difference equation
- Must include anti-windup

The reasoning for it to be implemented as a difference equation, is due to it being an exact relationship between current and past samples.

0.2 Deriving the PI difference equation

In order to obtain the difference equation for the PI-controller, the standard PI equation in the s-domain is observed:

$$u(s) = \left(k_p + \frac{k_i}{s}\right) \cdot e(s) \tag{0.1}$$

To get the PI equation into the time-discrete z-domain, it is chosen to use the Tustin's Method.
This is done by substiting s with the following

$$s = \frac{2}{T} \cdot \left(\frac{z-1}{z+1}\right)$$

Equation ref?? now becomes

$$u(z) = \left(k_p + k_i \cdot \frac{T}{2}\frac{z+1}{z-1}\right) \cdot e(z)$$

Now the expression on the right side is refactored

$$u(z) = \left(\frac{k_p \cdot 2(z-1) + k_i \cdot T \cdot (z+1)}{2 \cdot (z-1)}\right) \cdot e(z)$$

In order to have the discrete equation in terms of previous samples, both sides of the fraction are divided by z^{-1}

$$u(z) = \left(\frac{k_p \cdot (1-z^{-1}) + k_i \cdot \frac{T}{2} \cdot (1+z^{-1})}{(1-z^{-1})}\right) \cdot e(z)$$

It is wished to get an expressions in terms of u(z) and e(z) with no non-constant-fractions, therefore both sides are multiplied by $(1-z^{-1})$

$$u(z) \cdot (1-z^{-1}) = \left(k_p \cdot (z-1) + k_i \cdot \frac{T}{2} \cdot (z+1)\right) \cdot e(z)$$

The inverse z-transformation is now performed on both sides

$$u[n] - u[n-1] = k_p \cdot (e[n] - e[n-1]) + k_i \cdot \frac{T}{2}(e[n] + e[n-1])$$

The final difference equation becomes

$$u[n] = u[n-1] + k_p \cdot (e[n] - e[n-1]) + k_i \cdot \frac{T}{2}(e[n] + e[n-1])$$

0.3 Implementation of the difference equation

The design has been chosen.The difference equation has been derived. It is time to implement the PI-controller with anti-windup. The following code snippet will allow for anti-windup, whilst having the PI-controller expressed as a difference equation.

```
1
2
3  void calculate_PI(int16_t error, PI* K)
4  {
5      int16_t tmp_output = 0;
6      K->x_n[2] = K->x_n[1];
7      K->x_n[1] = K->x_n[0];
8      K->x_n[0] = error;
9
10     K->y_n[2] = K->y_n[1];
11     K->y_n[1] = K->y_n[0];
12     K->y_n[0] = K->y_n[1] + ( K->kp * (K->x_n[0] - K->x_n[1]) +
13     ( K->conditional_anti * K->ki * K->sample_rate / 2) * (K->x_n[0] + K->x_n[1]) );
14     K->conditional_anti = 1;
15
16     tmp_output = K->y_n[0];
17
18     if(tmp_output >= K->saturation)
19     {
20          tmp_output = K->saturation;
21          K->conditional_anti = 0;
22     }
23     else
24     {
25         if (tmp_output <= -K->saturation)
26         {
27             tmp_output = -K->saturation;
28             K->conditional_anti = 0;
29         }
30     }
```