**PID_REG3**

**Digital PID Controller with Anti-windup**

**Description**
This module implements a 32-bit digital PID controller with anti-windup correction. It can be used for PI or PD controller as well. In this digital PID controller, the differential equation is transformed to the difference equation by means of the backward approximation.

![Diagram of PID_REG3]

**Availability**
This IQ module is available in one interface format:

1) The C interface version

**Module Properties**
**Type:** Target Independent, Application Dependent

**Target Devices:** x281x or x280x

**C Version File Names:** pid_reg3.c, pid_reg3.h

**IQmath library files for C:** IQmathLib.h, IQmath.lib

<table>
<thead>
<tr>
<th>Item</th>
<th>C version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size(^1)</td>
<td>85/85 words</td>
<td></td>
</tr>
<tr>
<td>(x281x/x280x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data RAM</td>
<td>0 words(^*)</td>
<td></td>
</tr>
<tr>
<td>xDAIS ready</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>XDAIS component</td>
<td>No</td>
<td>IALG layer not implemented</td>
</tr>
<tr>
<td>Multiple instances</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Reentrancy</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

\(^*\) Each pre-initialized "._iq" PID_REG3 structure consumes 34 words in the data memory

\(^1\) Code size mentioned here is the size of the `calc()` function
The structure of PID_REG3 object is defined by following structure definition:

```c
typedef struct {
  _iq Ref;    // Input: Reference input
  _iq Fdb;    // Input: Feedback input
  _iq Err;    // Variable: Error
  _iq Kp;    // Parameter: Proportional gain
  _iq Up;    // Variable: Proportional output
  _iq Ui;    // Variable: Integral output
  _iq Ud;    // Variable: Derivative output
  _iq OutPreSat;    // Variable: Pre-saturated output
  _iq OutMax;    // Parameter: Maximum output
  _iq OutMin;    // Parameter: Minimum output
  _iq Out;    // Output: PID output
  _iq SatErr; // Variable: Saturated difference
  _iq Ki;    // Parameter: Integral gain
  _iq Kc;  // Parameter: Integral correction gain
  _iq Kd;  // Parameter: Derivative gain
  _iq Up1; // History: Previous proportional output
  void  (*calc)(); // Pointer to calculation function
} PIDREG3;
```

typedef PIDREG3 *PIDREG3_handle;

**Module Terminal Variables/Functions**

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>Description</th>
<th>Format</th>
<th>Range(Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Ref</td>
<td>Reference input</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Fdb</td>
<td>Feedback input</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>OutMax</td>
<td>Maximum PID32 module output</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>OutMin</td>
<td>Minimum PID32 module output</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td>Output</td>
<td>Out</td>
<td>PID Output (Saturated)</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td>PID parameter</td>
<td>Kp</td>
<td>Proportional gain</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Ki</td>
<td>Integral gain</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Kd</td>
<td>Derivative gain</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Kc</td>
<td>Integral correction gain</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td>Internal</td>
<td>Err</td>
<td>Error=Reference-feedback</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>SatErr</td>
<td>SatErr=output-preSatOut</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Up</td>
<td>Proportional output</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Up1</td>
<td>Previous proportional output</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Ui</td>
<td>Integral output</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Ud</td>
<td>Differential output</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>OutPreSat</td>
<td>PID output before saturation</td>
<td>GLOBAL_Q</td>
<td>80000000-7FFFFFFFFF</td>
</tr>
</tbody>
</table>

GLOBAL_Q valued between 1 and 30 is defined in the IQmathLib.h header file.
Special Constants and Data types

**PIDREG3**
The module definition is created as a data type. This makes it convenient to instance an interface to the PID module. To create multiple instances of the module simply declare variables of type PIDREG3.

**PIDREG3_handle**
User defined Data type of pointer to PID_REG3 module

**PIDREG3_DEFAULTS**
Structure symbolic constant to initialize PID_REG3 module. This provides the initial values to the terminal variables as well as method pointers.

Methods

```c
void pid_reg3_calc(PIDREG3_handle);
```

This function implements the digital PID controller (IQ implementation) using backward approximation technique. The input argument to this function is the module handle.

Module Usage

**Instantiation**
The following example instances two PID objects

```c
PIDREG3  pid1, pid2;
```

**Initialization**
To instance pre-initialized objects

```c
PIDREG3 pid1 = PIDREG3_DEFAULTS;
PIDREG3 pid2 = PIDREG3_DEFAULTS;
```

**Invoking the computation function**

```c
pid1.calc(&pid1);
pid2.calc(&pid2);
```

**Example**
The following pseudo code provides the information about the module usage.

```c
/* Instance the PID_REG3 module */
PIDREG3  pid1=PIDREG3_DEFAULTS;
PIDREG3  pid2=PIDREG3_DEFAULTS;
main()
{
    pid1.Kp = _IQ(0.5);  // Pass _iq parameters to pid1
    pid1.Ki  = _IQ(0.001); // Pass _iq parameters to pid1
    pid1.Kd = _IQ(0.01);  // Pass _iq parameters to pid1
    pid1.Kc = _IQ(0.9);   // Pass _iq parameters to pid1
```
void interrupt periodic_interrupt_isr()
{
    pid1.Ref = input1_1;  // Pass _iq inputs to pid1
    pid1.Fdb = input1_2;  // Pass _iq inputs to pid1
    pid2.Ref = input2_1;  // Pass _iq inputs to pid2
    pid2.Fdb = input2_2;  // Pass _iq inputs to pid2

    pid1.calc(&pid1);    // Call compute function for pid1
    pid2.calc(&pid2);    // Call compute function for pid2

    output1 = pid1.Out;  // Access the output of pid1
    output2 = pid2.Out;  // Access the output of pid2
}
Technical Background

The block diagram of a conventional PID controller with anti-windup correction can be shown in Figure 1.

\[
\begin{align*}
\text{Figure 1: Block diagram of PID controller with anti-windup}
\end{align*}
\]

The differential equation for PID controller with anti-windup before saturation is described in the following equation [1].

\[
\begin{align*}
\text{Equation (1):} \\
\text{Pre-saturated output:} \\
\text{Proportional term:} \quad u_{\text{presat}}(t) = u_p(t) + u_i(t) + u_d(t)\\
\end{align*}
\]

Each term can be expressed as follows:

- **Proportional term:**
  \[
  u_p(t) = K_p e(t) \tag{2}
  \]

- **Integral term with saturation correction:**
  \[
  u_i(t) = \frac{K_p}{T_i} \int_0^t e(\varsigma) d\varsigma + K_c(u(t) - u_{\text{presat}}(t)) \tag{3}
  \]

- **Derivative term:**
  \[
  u_d(t) = K_p T_d \frac{de(t)}{dt} \tag{4}
  \]

where
- \(u(t)\) is the output of PID controller
- \(u_{\text{presat}}(t)\) is the output before saturation
- \(e(t)\) is the error between the reference and feedback variables
- \(K_p\) is the proportional gain of PID controller
- \(T_i\) is the integral time (or reset time) of PID controller
- \(T_d\) is the derivative time of PID controller
- \(K_c\) is the integral correction gain of PID controller

Equations (1)-(4) can be discretized using backward approximation as follows:

- **Pre-saturated output:**
  \[
  u_{\text{presat}}(k) = u_p(k) + u_i(k) + u_d(k) \tag{5}
  \]

- **Proportional term:**
Technical Background

\[ u_p(k) = K_p e(k) \]  \hspace{1cm} (6)

Integral term with saturation correction:
\[ u_i(k) = u_i(k-1) + K_p \frac{T}{T_i} e(k) + K_c \left( u(k) - u_{\text{presat}}(k) \right) \]  \hspace{1cm} (7)

Derivative term:
\[ u_d(k) = K_p \frac{T_d}{T} \left( e(k) - e(k-1) \right) \]  \hspace{1cm} (8)

Defining \( K_i = \frac{T}{T_i} \), and \( K_d = \frac{T_d}{T} \), then integral with saturation correction and derivative terms finally become
\[ u_i(k) = u_i(k-1) + K_i u_p(k) + K_c \left( u(k) - u_{\text{presat}}(k) \right) \]  \hspace{1cm} (9)

\[ u_d(k) = K_d \left( u_p(k) - u_p(k-1) \right) \]  \hspace{1cm} (10)

where \( T \) is sampling period (sec).

Table 1 shows the correspondence of notation between variables used here and variables used in the program (i.e., pid_reg3.c and pid_reg3.h). The software module requires that both input and output variables are in per unit values.

<table>
<thead>
<tr>
<th></th>
<th>Equation Variables</th>
<th>Program Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td>Ref</td>
<td>Ref</td>
</tr>
<tr>
<td></td>
<td>Fdb</td>
<td>Fdb</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>u(k)</td>
<td>Out</td>
</tr>
<tr>
<td></td>
<td>e(k)</td>
<td>Err</td>
</tr>
<tr>
<td></td>
<td>u_p(k)</td>
<td>Up</td>
</tr>
<tr>
<td></td>
<td>u_p(k-1)</td>
<td>Up1</td>
</tr>
<tr>
<td></td>
<td>u_i(k)</td>
<td>Ui</td>
</tr>
<tr>
<td></td>
<td>u_d(k)</td>
<td>Ud</td>
</tr>
<tr>
<td></td>
<td>u_{\text{presat}}(k)</td>
<td>OutPreSat</td>
</tr>
<tr>
<td></td>
<td>e_{\text{sat}}(k)</td>
<td>SatErr</td>
</tr>
<tr>
<td></td>
<td>K_p</td>
<td>Kp</td>
</tr>
<tr>
<td></td>
<td>K_i</td>
<td>Ki</td>
</tr>
<tr>
<td></td>
<td>K_d</td>
<td>Kd</td>
</tr>
<tr>
<td></td>
<td>K_c</td>
<td>Kc</td>
</tr>
</tbody>
</table>

Table 1: Correspondence of notations

References: