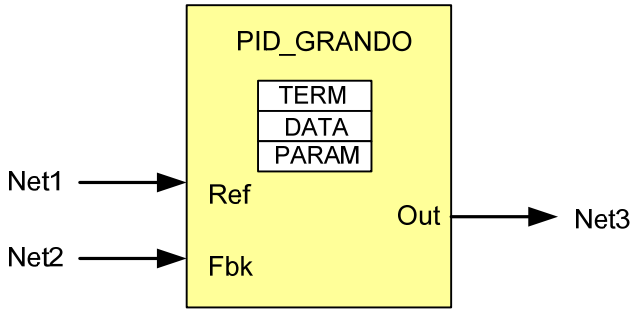
PID_GRANDO	*PID controller module*

This C macro implements a basic summing junction and PID control algorithm.



Macro file `PID_grando.h`

Module Summary

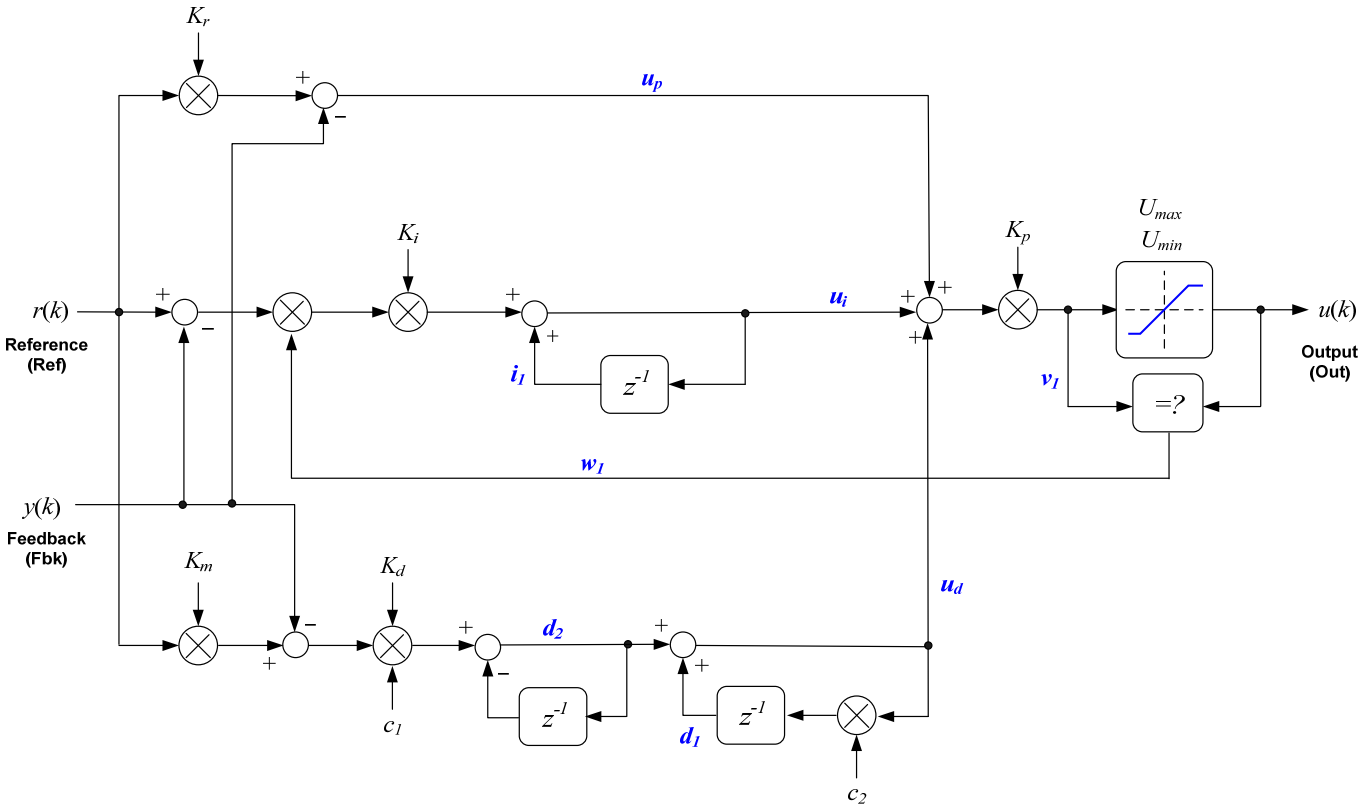
CPU Dependency	C28x
Device Dependency	None
Application	Digital control
C-based initialization	No
ASM interrupt initialization	Yes
ASM run-time macro	Yes
Multiple instanciation support	Yes
Re-entrant	No
Accessible from C	Yes
Full configuration from C	Yes
Input/Output format	Q24

General description

The PID_grando module implements a basic summing junction and PID control law with the following features:

- Programmable output saturation
- Independent reference weighting on proportional path
- Independent reference weighting on derivative path
- Anti-windup integrator reset
- Programmable derivative filter

All intput, output and internal data is in I8Q24 fixed-point format. A block diagram of the internal controller structure is shown below.



The code is supplied as a C macro in a single header file named "`PID_grando.h`". The controller variables are grouped into three short C structures as follows.

1. **Terminals**

Ref	// Input: reference set-point
Fdb	// Input: feedback
Out	// Output: controller output
c1	// Internal: derivative filter coefficient
c2	// Internal: derivative filter coefficient

2. **Parameters**

Kr	// Parameter: proportional reference
Kp	// Parameter: proportional loop gain
Ki	// Parameter: integral gain
Kd	// Parameter: derivative gain
Km	// Parameter: derivative reference weighting
Umax	// Parameter: upper saturation limit
Umin	// Parameter: lower saturation limit

3. **Data**

up	// Data: proportional term
ui	// Data: integral term
ud	// Data: derivative term
v1	// Data: pre-saturated controller output
i1	// Data: integrator storage: ui(k-1)
d1	// Data: differentiator storage: ud(k-1)
d2	// Data: differentiator storage: d2(k-1)
w1	// Data: saturation record: [u(k-1) - v(k-1)]

Technical description

a) *Proportional path*

The proportional term is taken as the difference between the reference and feedback terms. A feature of this controller is that sensitivity to the reference input can be weighted differently to the feedback path. This provides an extra degree of freedom when tuning the controller response to a dynamic input. The proportional law is:

$$u_p(k) = K_r r(k) - y(k) \quad \text{..... (1)}$$

Note that "proportional" gain is applied to the sum of all three terms and will be described in section d).

b) *Integral path*

The integral path consists of a discrete integrator which is pre-multiplied by a term derived from the output module. The term w1 is either zero or one, and provides a means to disable the integrator path when output saturation occurs. This prevents the integral term from "winding up" and improves the response time on recovery from saturation. The integrator law used is based on a backwards approximation.

$$u_i(k) = u_i(k-1) + K_i\left[r(k) - y(k)\right] \quad \text{..... (2)}$$

c) *Derivative path*

The derivative term is a backwards approximation of the difference between the current and previous inputs. The input is the difference between the reference and feedback terms, and like the proportional term, the reference path can be weighted independently to provide an additional variable for tuning.

A first order digital filter is applied to the derivative term to reduce nose amplification at high frequencies. Filter cutoff frequency is determined by two coefficients (c1 & c2). The derivative law is shown below.

$$e(k) = K_m r(k) - y(k) \quad \text{..... (3)}$$

$$u_d(k) = K_d\left[c_2 u_i(k-1) + c_1 e(k) - c_1 e(k-1)\right] \quad \text{..... (4)}$$

Filter coefficients are based on the cut-off frequency (a) and sample period (T) as follows:

$$c_1 = \frac{a}{1+aT} \quad \text{..... (5)}$$

$$c_2 = \frac{1}{1+aT} \quad \text{..... (6)}$$

d) *Output path*

The output path contains a multiplying term (Kp) which acts on the sum of the three controller parts. The result is then saturated according to user programmable upper and lower limits to give the output term.

The pre-and post-saturated terms are compared to determine whether saturation has occurred, and if so, a zero or one term is produced which is used to disable the integral path (see above). The output path law is defined as follows.

$$v_1(k) = K_p\left[u_p(k) + u_i(k) + u_d(k)\right] \quad \text{..... (7)}$$

$$u(k) = \begin{cases} U_{\max} : v_1(k) > U_{\max} \\ U_{\min} : v_1(k) < U_{\min} \\ v_1(k) : U_{\min} < v_1(k) < U_{\max} \end{cases} \quad \text{..... (8)}$$

$$w_1(k) = \begin{cases} 0 : v_1(k) \neq u(k) \\ 1 : v_1(k) = u(k) \end{cases} \quad \text{..... (9)}$$

Tuning the controller

Default values for the controller coefficients are defined in the macro header file which apply unity gain through the proportional path, and disable both integral and derivative paths. A suggested general technique for tuning the controller is now described.

Steps 1-4 are based on tuning a transient produced either by a step load change or a set-point step change.

Step 1. Ensure integral and derivative gains are set to zero. Ensure also the reference weighting coefficients (K_r & K_m) are set to one.

Step 2. Gradually adjust proportional gain variable (K_p) while observing the step response to achieve optimum rise time and overshoot compromise.

Step 3. If necessary, gradually increase integral gain (K_i) to optimize the return of the steady state output to nominal. This will probably be accompanied by an increase in overshoot and oscillation, so it may be necessary to slightly decrease the K_p term again to find the best balance.

Step 4. If the transient response exhibits excessive oscillation, this can sometimes be reduced by applying a small amount of derivative gain. To do this, first ensure the coefficients c_1 & c_2 are set to one and zero respectively. Next, slowly add a small amount of derivative gain (K_d). The controller will be very sensitive to this term and may become unstable so be sure to start with a very small number.

Steps 5 & 6 only apply in the case of tuning a transient set-point. In the regulator case, or where the set-point is fixed and tuning is conducted against changing load conditions, they are not useful.

Step 5. Overshoot and oscillation following a set-point transient can sometimes be improved by lowering the reference weighting in the proportional path. To do this, gradually reduce the K_r term from its nominal unity value to optimize the transient. Note that this will change the loop sensitivity to the input reference, so the steady state condition will change unless integral gain is used.

Step 6. If derivative gain has been applied, transient response can often be improved by changing the reference weighting, in the same way as step 6 except that in the derivative case steady state should not be affected. Slowly reduce the K_m variable from it's nominal unity value to optimize overshoot and oscillation. Note that in many cases optimal performance is achieved with a reference weight of zero in the derivative path, meaning that the differential term acts on purely the output, with no contribution from the input reference.

The derivative path introduces a term which has a frequency dependent gain. At higher frequencies, this can cause noise amplification in the loop which may degrade servo performance. If this is the case, it is possible to filter the derivative term using a first order digital filter in the derivative path. Steps 7 & 8 describe the derivative filter.

Step 7. Select a filter roll-off frequency in radians/second. Use this in conjunction with the system sample period (T) to calculate the filter coefficients c_1 & c_2 (see equations 5 & 6).

Step 8. Note that the c_1 coefficient will change the derivative path gain, so adjust the value of K_d to compensate for the filter gain. Repeat steps 5 & 6 to optimize derivative path gain.

Code listing

```
/* ------------------------------------------------------------------------------
File name:     PID_grando.h

Originator:    C2000 System Applications, Texas Instruments

Description:   Data and macro definitions for "grando" PID controller
------------------------------------------------------------------------------*/

#ifndef __PID_GRANDO_H__
#define __PID_GRANDO_H__

typedef struct {  _iq  Ref;                         // Input: reference set-point
                  _iq  Fbk;                         // Input: feedback
                  _iq  Out;                         // Output: controller output
                  _iq  c1;                          // Internal: derivative filter coefficient 1
                  _iq  c2;                          // Internal: derivative filter coefficient 2
                } PID_GRANDO_TERMINALS;
                // note: c1 & c2 placed here to keep structure size under 8 words

typedef struct {  _iq  Kr;                          // Parameter: reference set-point weighting
                  _iq  Kp;                          // Parameter: proportional loop gain
                  _iq  Ki;                          // Parameter: integral gain
                  _iq  Kd;                          // Parameter: derivative gain
                  _iq  Km;                          // Parameter: derivative weighting
                  _iq  Umax;                        // Parameter: upper saturation limit
                  _iq  Umin;                        // Parameter: lower saturation limit
                } PID_GRANDO_PARAMETERS;

typedef struct {  _iq  up;                          // Data: proportional term
                  _iq  ui;                          // Data: integral term
                  _iq  ud;                          // Data: derivative term
                  _iq  v1;                          // Data: pre-saturated controller output
                  _iq  i1;                          // Data: integrator storage: ui(k-1)
                  _iq  d1;                          // Data: differentiator storage: ud(k-1)
                  _iq  d2;                          // Data: differentiator storage: d2(k-1)
                  _iq  w1;                          // Data: saturation record: [u(k-1) - v(k-1)]
                } PID_GRANDO_DATA;


typedef struct {  PID_GRANDO_TERMINALS       term;
                  PID_GRANDO_PARAMETERS      param;
                  PID_GRANDO_DATA            data;
               } PID_GRANDO_CONTROLLER;


typedef PID_GRANDO_CONTROLLER *PID_handle;


/*-----------------------------------------------------------------------------
Default initalisation values for the PID_GRANDO objects
-----------------------------------------------------------------------------*/

#define PID_TERM_DEFAULTS {                         \
                          0,                        \
                          0,                        \
                          0,                        \
                          0,                        \
                          0                         \
                      }

#define PID_PARAM_DEFAULTS {          \
                         _IQ(1.0),  \
                         _IQ(1.0),  \
                         _IQ(0.0),  \
                         _IQ(0.0),  \
                         _IQ(1.0),  \
                         _IQ(1.0),  \
                         _IQ(-1.0)  \
                            }

#define PID_DATA_DEFAULTS {           \
                         _IQ(0.0),  \
                         _IQ(0.0),  \
                         _IQ(0.0),  \
                         _IQ(0.0),  \
                         _IQ(0.0),  \
```

```
                        _IQ(0.0),  \
                        _IQ(0.0),  \
                        _IQ(1.0)   \
                             }


/*-----------------------------------------------------------------------------
       PID_GRANDO Macro Definition
-----------------------------------------------------------------------------*/

#define PID_GR_MACRO(v)                                                        \
                                                                               \
                                                                               \
       /* proportional term */                                                 \
       v.data.up = _IQmpy(v.param.Kr, v.term.Ref) - v.term.Fbk;                \
                                                                               \
       /* integral term */                                                     \
       v.data.ui = _IQmpy(v.param.Ki, _IQmpy(v.data.w1, (v.term.Ref - v.term.Fbk))) + v.data.i1; \
       v.data.i1 = v.data.ui;                                                  \
                                                                               \
       /* derivative term */                                                   \
       v.data.d2 = _IQmpy(v.param.Kd, _IQmpy(v.term.c1, (_IQmpy(v.term.Ref, v.param.Km) - v.term.Fbk))) -
v.data.d2;     \
       v.data.ud = v.data.d2 + v.data.d1;                                      \
       v.data.d1 = _IQmpy(v.data.ud, v.term.c2);                               \
                                                                               \
       /* control output */                                                    \
       v.data.v1 = _IQmpy(v.param.Kp, (v.data.up + v.data.ui + v.data.ud));    \
       v.term.Out= _IQsat(v.data.v1, v.param.Umax, v.param.Umin);               \
       v.data.w1 = (v.term.Out == v.data.v1) ? _IQ(1.0) : _IQ(0.0);

#endif // __PID_GRANDO_H__
```