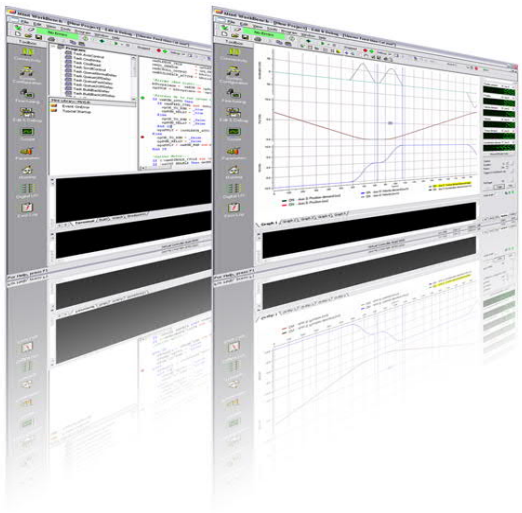
Motion Control Products

Application note
PID Controller

AN00208

Rev B (EN)

Use the flexibility of Mint to design and code your own custom PID algorithms for applications such as unwinding, rewinding and temperature control



Introduction

ABB servo products implement PID style control loops when processing torque, velocity and position control of motion axes. These algorithms use the drive current or motor position/velocity as feedback. These are sufficient if you only want to control these basic axis parameters.

On some ABB servo products an additional motion function is available called “Hold To Analog” or “HTA”. HTA is effectively a PID (Proportional, Integral, and Derivative) controller that allows an axis *position* to be automatically adjusted such that it tries to maintain an analog input at a pre-programmed setpoint / level. It could be used to control the height/position of a Z axis on a plasma or laser cutting machine for example, where the actual height above the material surface is fed back via an analog input.

There are times when we may need to control some other process variable which is not related to positional movement of an axis. We might want to control an analogue output based on the value of an analogue input or control the speed/torque of an axis in order to try and maintain an analog input at a preset value for example. Alternatively, we may be controlling an axis in a HTA style but need more control / flexibility than the standard integrated HTA algorithm. For these purposes this application note describes how to implement a custom PID control algorithm in Mint.

How PID control works

This is a short summary of the function of a PID controller. It is beyond the scope of this document to describe a detailed account of how a PID controller should be setup. There is plenty of literature freely available for this.

A PID controller calculates an "error" value as the difference between a measured (input) process variable and a desired set-point. The controller attempts to minimize the error by adjusting the process control output which should in turn affect the measured (input) process variable.

The PID controller algorithm involves three separate constant gain parameters: the proportional, the integral and derivative values, denoted Kp, Ki, and Kd. These gains track how the error varies with time. Kp depends on the *present* error, Ki on the accumulation of *past* errors, and Kd is a prediction of *future* errors, based on the current rate of change. The sum of these three components is used to adjust the process via a process control output.

The *proportional* term produces an output value that is proportional to the current error value. A high proportional gain results in a large change in the output for a given change in the error. If the proportional gain is too high, the system can become unstable. The proportional term will usually make up the bulk of the output change.

The *integral* term is the sum of errors over time. The integral term eliminates the residual steady state error that occurs with a purely proportional controller. It is also good at accelerating the process variable towards the set-point. If it is set too high it can cause the process variable to overshoot the set-point or oscillate around it.

The *derivative* term counteracts any rapid changes in the error. It reduces the overshoot produced by the integral term. A high setting will reduce the responsiveness of the controller. Care should be taken if the feedback signal is noisy as the derivative term will amplify the noise to a greater extent than the other gain terms.

Coding a PID controller in Mint

Example 1: System requiring a process control output even when the error is zero (e.g. winding or unwinding a reel and maintaining constant tension)

In order to make the code more re-usable the data for the PID controller is saved in a “Structure” variable type. The Structure variable is passed to the doPID function each time it is used. The following structure definition should be placed at the top of the program.

```
Structure TPID
  'Setup parameters
  PIDMaxOut As Float
  PIDMinOut As Float
  PIDKP As Float
  PIDKI  As Float
  PIDKILimit As Float
  PIDKD As Float
  PIDLoopTime As Float
  'Internal variables
  tTime As Time
  fError As Float
  fDerivative As Float
  fErrorLast As Float
  fIntegral As Float
  fUnlimitedPidOut As Float
End Structure
```

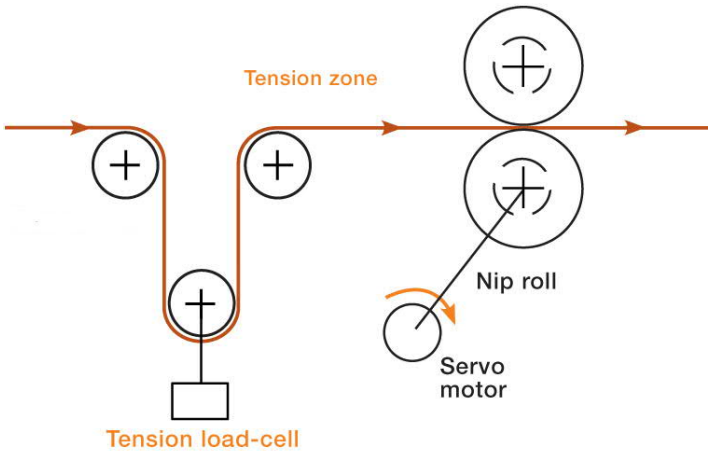
Now the doPID function should be placed in the program:

```
Function doPID(ByRef PIDin As TPID, ByVal fTarget, ByVal fFeedback, ByVal fCurrentVal) As Float
  Dim fKIntLimit

  'Wait for servo loop timer to expire
  Pause PIDin.tTime > PIDin.PIDLoopTime
  PIDin.tTime = 0

  'Calculate PID error
  PIDin.fError =  fTarget - fFeedback

  'Calculate new integral value and limit integral affect
  If Sgn(PIDIn.fError) <> Sgn(PIDIn.fErrorLast) Then PIDin.fIntegral = 0
  PIDin.fIntegral = PIDin.fIntegral + PIDin.fError
  fKintLimit = ((hmiPID_KILIMIT / 100) * PIDin.PIDMaxOut)
  If (PIDin.fError > 0) And (PIDin.fIntegral > fKIntLimit) Then PIDin.fIntegral = fKIntLimit
  If (PIDin.fError < 0) And (PIDin.fIntegral < -fKIntLimit) Then PIDin.fIntegral = -fKIntLimit
```

```
'Calculate derivative value
PIDin.fDerivative = PIDin.fError - PIDin.fErrorLast
PIDin.fErrorLast = PIDin.fError

'Calculate new PID output
PIDin.fUnlimitedPidOut = (PIDin.PIDKP * PIDin.fError) + _
                         (PIDin.PIDKI * PIDin.fIntegral) + _
                         (PIDin.PIDKD * PIDin.fDerivative) + _
                         fCurrentVal   'Include this line for PID modes other than position

'Range check PID output
If (PIDin.fUnlimitedPidOut > PIDin.PIDMaxOut ) Then
  doPID = PIDin.PIDMaxOut
ElseIf (PIDin.fUnlimitedPidOut < PIDin.PIDMinOut) Then
  doPID = PIDin.PIDMinOut
Else
  doPID = PIDin.fUnlimitedPidOut
EndIf

End Function
```

The example function adds the effect of the PID gains to the existing process output value. This form of the code would be used, for example, in systems where the axis needs to keep applying a *speed* (or *torque*) to maintain the process input at the desired value. For systems where the axis *position* changes to control the process the PID output might be expected to reduce to zero once the required position is achieved so there would be no need to include the current process output value in these cases.

In the diagram opposite we are controlling the tension in a web by applying a torque reference to a servo running a nip roll. If we used the servo current control loop on its own we could achieve good open loop tension control. We would achieve a better result though if we could get some feedback from the achieved/actual tension.



Closed loop control would use a dancer arm or load cell to measure the tension in the web. We make use of the fact that tension at the nip roll is the product of motor torque and the radius of the nip roll. In this case the nip roll radius is fixed and the motor has a fixed torque constant. By taking into account gearing between the motor and the nip roll we can derive a value for a scaling constant to convert between motor torque reference and material tension (and vice versa).

We read tension feedback from the load cell via an analogue input to the motion controller (or intelligent drive) which gives us a reading from 0 to 10Vdc (0 to 100% in Mint).

We need to define the analogue input used to read the load cell tension and a scaling factor for this to convert to Newtons of force/tension. We need a constant for our fixed nip roll radius and another value for the motor torque constant (taking into account any intermediate gearing). Variables are also included to define the tension set-point and the process output variable nip tension.

```
Const _fLoadCellScale As Float = 0.5 '0-100% = 0-50N
Const _fNipRollRadius As Float = 0.1 'm
Const _fMotorTorqueConstant As Float = 3.5  'Nm/A

Define aiLoadCell = (ADC(0) * _fLoadCellScale)
```

```
Dim fTensionSP As Float = 30 'N
Dim fNipTensionDemand As Float
Dim fPIDOutput As Float
```

Our PID data structure allows us to set some gains (which we can fine tune as necessary during application development) and limits for the integral term.

```
TensionPID.PidKProp     = 1.0
TensionPID.PidKInt      = 0.05
TensionPID.PidKIntLimit = 15
TensionPID.PidKDeriv    = 0.05
```

The PID controller is then called repeatedly in a loop. The output of the PID loop is used to apply the torque reference:

```
Loop
  fNipTensionDemand = (DRIVERATEDCURRENT * _fMotorTorqueConstant * TorqueRef(_axNip) )/ (100 * _fNipRollRadius))
  fPIDOutput = doPID( TensionPID, fTensionSP, aiLoadCell, fNipTensionDemand )
  TorqueRef( _axNip ) = 100 * (fPIDOutput * _fNipRollRadius) / (_fMotorTorqueConstant * DRIVERATEDCURRENT)
End Loop
```

Note that the Mint keyword DRIVERATEDCURRENT is only available on intelligent drives so for tension control systems using a motion controller this keyword would be replaced by another hard coded constant instead.

An example Mint program is included with this application note (Tension PID Control.mnt). For simplicity this uses a single constant that encompasses all of the system variables (torque constant, nip roll radius, drive rated current) detailed above. The user may wish to expand this, as shown above, in cases where it needs to be made clear how the torque reference is derived. It is also possible to enhance the functionality of the example code to include elements to compensate for more complex factors such as:

- Reel diameter compensation or taper tension schemes
- Inertia compensation
- Friction compensation

Example 2: System requiring no process control output once the error is zero (e.g. positioning an axis – once the axis is in position the speed reference becomes zero)

This example is similar in operation to the Mint HTA function, but whereas HTA requires an analog input as feedback this example uses the axis position itself as the process feedback. In the example the system receives the “demand position” via NETFLOAT(0). The code scales this demand position (which is in the range 0-255) so that the actual position achieved is in the range 0 to 90 user units. It then uses a PID loop to generate a speed/velocity reference that moves the axis to the required position.

Limits on the maximum and minimum velocity reference are set via member variables of the PID structure…

```
PositionPID.PIDMaxOut   = 50  'vel units/sec
PositionPID.PIDMinOut   = -50 'vel units/sec
```

As with Example 1, a “Structure” variable type is used for the PID controller. The structure variable is then passed to the doPID function each time it is used. Because we no longer need to maintain a process output once the error is zero there is no need to pass the current process output value to the doPID function so the example uses a slightly different form of this function…

```
Function doPID(ByRef PIDin As TPID, ByVal fTarget, ByVal fFeedback) As Float
  Dim fKIntLimit

  'Wait for servo loop timer to expire
  Pause PIDin.tTime > PIDin.PIDLoopTime
  PIDin.tTime = 0
```

```
  'Calculate PID error
  PIDin.fError =  fTarget - fFeedback

  'Calculate new integral value and limit integral affect
  If Sgn(PIDIn.fError) <> Sgn(PIDIn.fErrorLast) Then PIDin.fIntegral = 0
  PIDin.fIntegral = PIDin.fIntegral + PIDin.fError
  fKintLimit = ((hmiPID_KILIMIT / 100) * PIDin.PIDMaxOut)
  If (PIDin.fError > 0) And (PIDin.fIntegral > fKIntLimit) Then PIDin.fIntegral = fKIntLimit
  If (PIDin.fError < 0) And (PIDin.fIntegral < -fKIntLimit) Then PIDin.fIntegral = -fKIntLimit

  'Calculate derivative value
  PIDin.fDerivative = PIDin.fError - PIDin.fErrorLast
  PIDin.fErrorLast = PIDin.fError

  'Calculate new PID output
  PIDin.fUnlimitedPidOut = (PIDin.PIDKP * PIDin.fError) + _
                           (PIDin.PIDKI * PIDin.fIntegral) + _
                           (PIDin.PIDKD * PIDin.fDerivative)

  'Range check PID output
  If (PIDin.fUnlimitedPidOut > PIDin.PIDMaxOut ) Then
    doPID = PIDin.PIDMaxOut
  ElseIf (PIDin.fUnlimitedPidOut < PIDin.PIDMinOut) Then
    doPID = PIDin.PIDMinOut
  Else
    doPID = PIDin.fUnlimitedPidOut
  EndIf

End Function
```

The main program loop simply passes the PID structure, the demand position (scaled to user units) and measured position to the function…

```
Loop
  If ipStart Then
    'Update PID controller gain values from HMI
    PositionPID.PIDKP      = hmiPID_KP
    PositionPID.PIDKI      = hmiPID_KI
    PositionPID.PIDKILimit = hmiPID_KILIMIT
    PositionPID.PIDKD      = hmiPID_KD
    'Use a PID loop to adjust the target axis velocity reference
    VELREF(_axX) = doPID(PositionPID, cmPOS_DEMAND * _fPositionScale, POS(_axX))
  Else
    CANCEL(_axX)
    Pause(IDLE(_axX))
  End If
End Loop
```

The user just needs to adjust the limits on the maximum and minimum process outputs (in this case velocity reference), the acceleration and deceleration rates of the axis and the PID loop proportional, integral and derivative terms to achieve the required response. An example Mint program (Position PID Control.mnt) is included with this application note for reference.

Contact us

For more information please contact your
local ABB representative or one of the following:

new.abb.com/drives/low-voltage-ac/motion
new.abb.com/drives
new.abb.com/channel-partners
new.abb.com/plc