Perform Common Process Loop
Control Algorithms

Using the PIDE Instruction

Perform Common Process Loop Control Algorithms

Introduction

This white paper discusses how to use the features inherent in the Enhanced PIDE instruction in the RSLogix 5000 Function Block Diagram (FBD) editor to perform common process loop control algorithms such as:

- adaptive gains
- cascade control
- ratio control
- multiloop selection
- split-range time-proportioning

Although this paper focuses on the PIDE instruction, be aware that the FBD editor supports many other process control instructions. Other instructions provide capabilities such as flow totalization, ramp/soak temperature profiles, motor operated valve control, and two- or three-state device control for devices such as pumps and solenoid valves. These instructions provide you with the building blocks you need to perform typical process control applications.

Velocity vs. Positional Control

The PIDE instruction uses a velocity form algorithm of the PID equation. Essentially, this means that the loop works on change in error to change the output. Traditional PID algorithms used in PLCs have used positional form algorithms. A positional form algorithm works on error directly. Although this is acceptable for simple applications, the velocity form algorithm is much easier to apply for more advanced applications such as adaptive gains or multiloop selection. For this reason, most Distributed Control Systems (DCS) have traditionally used a velocity form algorithm. Likewise, the Logix controller family also takes advantage of the more advanced properties of a velocity form algorithm.

Understand that both a positional form and a velocity form PID algorithm perform identically in response to a change in error. In fact, you can easily derive one form of the equation from the other. The equations for the two types of algorithms are shown below:

Positional Form PID Algorithm

$$CV = K_P E + \sum K_I E \Delta t + K_D \frac{\Delta E}{\Delta t}$$

Velocity Form PID Algorithm

$$CV_n = CV_{n-1} + K_P \Delta E + K_I E \Delta t + K_D \frac{E_n - 2E_{n-1} + E_{n-2}}{\Delta t}$$

where:
CV = Controlled Variable
E = Error
Δt = Update time
K_p = Proportional gain
K_I = Integral gain
K_D = Derivative gain

The two main differences between the forms of the PID algorithm are that:

- the proportional term works on change in error (ΔE) in the velocity form and on error (E) in the positional form

- the accumulation of the integral term is contained in the previous output (CV_{n-1}) in the velocity form and in the summation of the integral term in the positional form. The following sections explain why this is important.

The PIDE instruction also supports two different forms of the velocity form algorithm – independent and dependent gains. These are described below:

Independent Gains Form
In this form of the algorithm, each term of the algorithm, proportional, integral, and derivative, has a separate gain. Changing one gain affects only that term and not any of the others.

$$CV_n = CV_{n-1} + K_P \Delta E + \frac{K_I}{60} E \Delta t + 60 K_D \frac{E_n - 2E_{n-1} + E_{n-2}}{\Delta t}$$

where:
CV = Control variable
E = Error in percent of span
Δt = Update time in seconds used by the loop
K_P = Proportional gain
K_I = Integral gain in min^{-1}. Note that a larger value of K_I causes a faster integral response.
K_D = Derivative gain in minutes

Dependent Gains Form

In this form of the algorithm, the Proportional gain is effectively changed into a Controller gain. By changing the Controller gain, you change the action of all three terms, proportional, integral, and derivative, at the same time.

$$CV_n = CV_{n-1} + K_C\left(\Delta E + \frac{1}{60T_I}E\Delta t + 60T_D\frac{E_n - 2E_{n-1} + E_{n-2}}{\Delta t}\right)$$

where:

CV = Control variable

E = Error in percent of span

Δt = Update time in seconds used by the loop

K_C = Controller gain

T_I = Integral time constant in minutes per repeat. In other words, it will take T_I minutes for the integral term to repeat the action of the proportional term in response to a step change in error. Note that a larger value of T_I causes a slower integral response.

T_D = Derivative time constant in minutes

When you use the PIDE instruction with the parameter DependIndepend cleared, the parameters PGain, IGain, and DGain are used to represent K_P, K_I, and K_D. When DependIndepend is set, you use the parameters PGain, IGain, and DGain to represent K_C, T_I, and T_D.

The PIDE equations above are representative of the algorithms used by the PIDE instruction. You can substitute the change in error values by the change in PV (in percent of span) for the proportional and derivative terms by manipulating the parameters PVEProportional and PVEDerivative. By default, the PIDE instruction actually uses the change in error for the proportional term and the change in PV for the derivative term. This eliminates large derivative spikes on changes in setpoint.

You can convert the gains used between the independent and dependent gains PIDE algorithm forms by using the following equations:

- $$K_P = K_C$$

- $$K_I = \frac{K_C}{T_I}$$

- $$K_D = K_C T_D$$

Either algorithm type – independent or dependent – can give you identical control with the appropriate gains. It really just depends on with which style of PID algorithm you have the most familiarity. Some people prefer the independent gains style since they can manipulate individual gains without affecting the other terms. Others prefer the dependent gains style since they can, at least to a certain extent, change only the controller gain and cause an overall change in the aggressiveness of the PID loop without changing each gain separately.
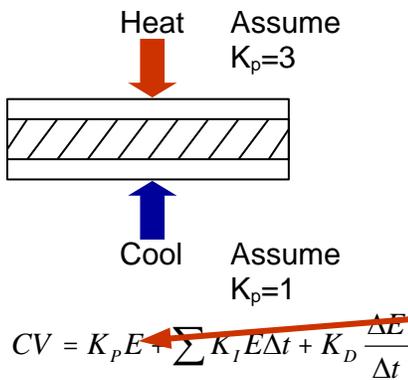
Adaptive Gains

One of the big advantages of a velocity form algorithm is the implementation of adaptive gains. Implementing adaptive gains simply means that you change the proportional, integral, and derivative gain values in a running loop. This is often desirable since a process may have very different operating characteristics depending on the actual operating environment. For example, the barrel temperature control of an extrusion machine often involves heating the barrel with resistive heaters and cooling the barrel by running coolant through lines around the barrel. The heating and cooling of the barrel are two different physical processes and often require different gain values in order to obtain the best control. Typically this is accomplished by defining 50% loop output as providing no heating or cooling. An output greater than 50% applies increasing heating, and an output less than 50 % applies increasing cooling.

With a positional form algorithm, swapping in new gains as the loop changes from heating to cooling is very difficult. Since the proportional term on a positional form algorithm works directly on error, any error at the point at which the gains change causes a bump in output proportional to the difference between the heating and cooling proportional gains. For example, assume that the heating proportional gain is 3 and the cooling proportional gain is 1. Now, if the loop output moves from cooling to heating (i.e., crosses 50%), and, when it does

so, the error is at 5%, the loop experiences a 10% bump in output when you switch it to the heating gains. This is because the positional form PID algorithm works on error directly. It is very difficult to get good control when the output is bumped just because of new gains. To handle this correctly, you really need to put the loop in manual, put in the new gains, execute the loop in manual once to allow it to back-calculate for a bumpless transfer, and then finally put the loop back into automatic to start using the new gains. This is difficult and time-consuming to program correctly.

Now do the same thing with a velocity form algorithm. Changing the proportional gain from 1 to 3 with a 5% error causes no change in the output since the error was 5% just before the new gain was used and remained 5% just after the new gain was used. Since the error didn't change, no proportional term change is made to the output. You now see why implementing adaptive gains control schemes is much easier with the velocity form PIDE algorithm used in Logix. You can simply swap in new gain values on-the-fly without worrying about bumping your process.

Heat   Assume K_p=3

Cool   Assume K_p=1

$$CV = K_P E + \sum K_I E \Delta t + K_D \frac{\Delta E}{\Delta t}$$

$$CV_n = CV_{n-1} + K_P \Delta E + K_I E \Delta t + K_D \frac{E_n - 2E_{n-1} + E_{n-2}}{\Delta t}$$

- **Assume that when we switch from cool to heat the error is 5%.**

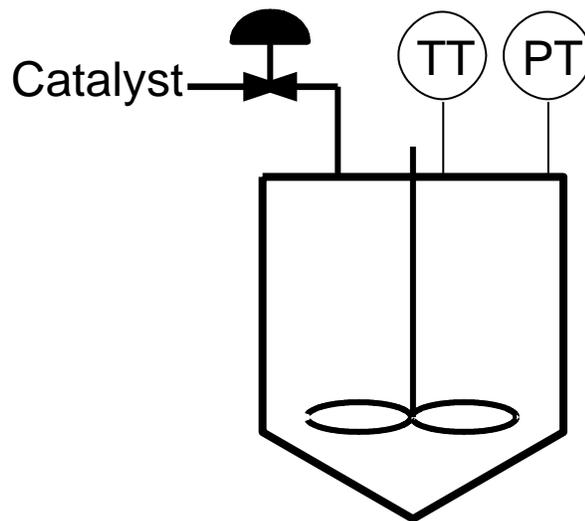- **Positional algorithm has an immediate 10% bump in output.**

- **Velocity algorithm has no bump in output.**

Multiloop Selection

There are times when two or more process variables are controlled by the same control variable. Often, the actual control variable sent to the field actuator needs to be limited in these cases to use either the lesser or greater of the outputs of two or more PID loops (one for each different process variable). For example, to control both the temperature and pressure in an exothermic chemical reaction, you

might have a PID loop for temperature, another PID loop for pressure, and use the lesser of the outputs of these two loops to control a flow of catalyst into the reactor to modulate the reaction rate. In other words, if the pressure is too high, the pressure PID loop calls for less catalyst, and if instead the temperature is too high, the temperature loop calls for less catalyst. In either case, you always want to use the lesser of the two loops to control the catalyst flow. The challenge is to align the loop which is not in control with the loop that is, and to allow control to bumplessly switch between the loops.
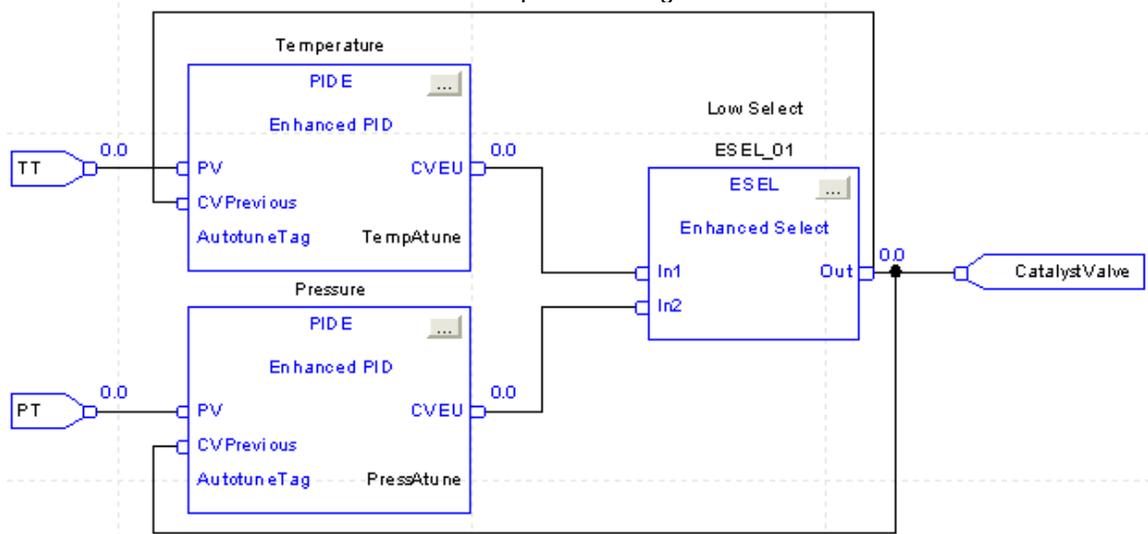
Catalyst — TT PT

A positional form PID algorithm is very difficult to implement correctly for this type of control scheme. Since you constantly need to align the loops, you must take the output of the loop which is in control, provide it as a manual output signal to the other loop, put that loop in manual to back calculate and align with the loop which is in control, and then put the out-of-control loop back into auto. You must do this for every

execution of the loops.

A velocity form PID algorithm provides a clear advantage for these types of control schemes. Since the previous output of the loop is available in the CV_{n-1} term, it is a simple matter of wiring the output actually sent to the final control element into the CV_{n-1} term of each loop. The two loops will therefore always be aligned with each other and control can bumplessly move between temperature or pressure limited control. An example of this logic is shown below:

As shown, the value actually sent to the catalyst valve is simply wired back into the CVPrevious parameter on each PIDE instruction. You should also set the CVSetPrevious parameter to tell the PIDE instruction to use the CVPrevious parameter as the CV_{n-1} term in the PID algorithm. This logic is much simpler to create and maintain than the equivalent logic required for a positional form PID algorithm.

Mode Control Options

The PIDE instruction provides additional capabilities through the use of many different modes of control. In addition to the traditional modes such as auto and manual, the PIDE instruction also supports the concept of Program/Operator control to define who is allowed to make changes to the loop. If the loop is in Program control, the user program can place the loop into the appropriate mode (e.g., Auto/Manual), and change the setpoint or manual output of the loop. Conversely, if the loop is in Operator control, the operator can change modes and values. The supported control types and loop modes are:

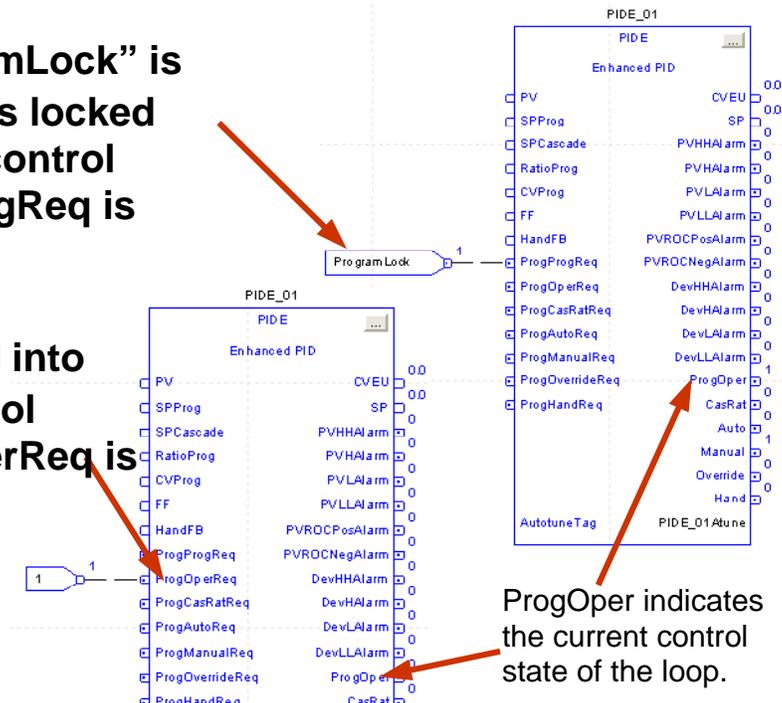| Mode	Usage
Program Control	When in Program control, the loop mode is determined by the user program. The user program can also change the setpoint and manual output of the loop.
Operator Control	When in Operator control, the loop mode is determined by the operator. The operator can also change the setpoint and manual output of the loop.
Cascade/Ratio Mode	When in Cascade/Ratio mode, the loop will automatically regulate its output as in Auto mode, but the setpoint will come from an external source, as connected to the SPCascade input parameter. If the UseRatio parameter is set, the SPCascade input will also be multiplied by a ratio value before it is used as the setpoint.
Auto Mode	When in Auto mode, the loop will automatically regulate its output to maintain the PV at the setpoint.
Manual Mode	When in Manual mode, the loop will set its output equal to the CV value entered by the user program (when in Program control) or by the operator (when in Operator control)
Override Mode	When in Override mode, the loop will set its output equal to the CV value configured in the CVOverride parameter. Override mode is typically used for interlock conditions.
Hand Mode	When in Hand mode, the CV value is set equal to the HandFB parameter. The HandFB is intended to come from a hard hand/auto station. When the loop is placed into Hand mode (by setting the ProgHandReq parameter), it indicates that the hand/auto station has bypassed the control system and is controlling the final control element directly. By setting the CV equal to the HandFB value (the output of the hand/auto station), the loop can bumplessly return to Auto or Manual mode once out of Hand mode.

Mode changes are initiated by setting the appropriate mode request parameters of the PIDE instruction. These mode request parameters are prefixed by either "Prog" to indicate it is a programmatic request or by "Oper" to indicate it is an operator generated request. For example, OperAutoReq is a request from the operator to enter Auto mode.

The Program/Operator control states can be used to lock the PIDE instruction into the appropriate control state when needed. For example, an automated startup sequence might be used in an application where the user program needs to lock the PIDE instruction into Program control to ensure that the operator does not interfere with the startup. This can be done by setting the ProgProgReq parameter (programmatic request to go to the Program control state).

- **When "ProgramLock" is true, the loop is locked into Program control since ProgProgReq is true.**

- **Loop is locked into Operator control since ProgOperReq is true.**

PIDE_01

PIDE

Enhanced PID

PV	CVEU	0.0
SPProg	SP	0.0
SPCascade	PVHHAlarm	0
RatioProg	PVHAlarm	0
CVProg	PVLAlarm	0
FF	PVLLAlarm	0
HandFB	PVROCPosAlarm	0
Program Lock — 1 — ProgProgReq	PVROCNegAlarm	0
ProgOperReq	DevHHAlarm	0
ProgCasRatReq	DevHAlarm	0
ProgAutoReq	DevLAlarm	0
ProgManualReq	DevLLAlarm	0
ProgOverrideReq	ProgOper	1
ProgHandReq	CasRat	0
	Auto	1
	Manual	1
	Override	0
	Hand	0
AutotuneTag	PIDE_01Atune	

PIDE_01

PIDE

Enhanced PID

PV	CVEU	0.0
SPProg	SP	0.0
SPCascade	PVHHAlarm	0
RatioProg	PVHAlarm	0
CVProg	PVLAlarm	0
FF	PVLLAlarm	0
HandFB	PVROCPosAlarm	0
ProgProgReq	PVROCNegAlarm	0
1 — 1 — ProgOperReq	DevHHAlarm	0
ProgCasRatReq	DevHAlarm	0
ProgAutoReq	DevLAlarm	0
ProgManualReq	DevLLAlarm	0
ProgOverrideReq	ProgOper	0
ProgHandReq	CasRat	

ProgOper indicates the current control state of the loop.

The control states and modes also have precedences. If both Program and Operator control are requested, the loop will go to Operator control. The precedence for modes are (lowest to highest): Cascade/Ratio, Auto, Manual, Override, and Hand.

Finally, the Operator requests are designed to simplify working with operator interfaces. Any time an "Oper…" request is set, the PIDE instruction evaluates whether it can respond to the request, and then always resets the request. This eliminates the need for special programming in the HMI to reset mode requests.

Timing Modes

Regulatory control instructions require a known update time (the Δt in the PIDE equation, for example) in order to execute correctly. In Logix, instructions such as Enhanced PID, Totalizer, and Lead-Lag support three different timing modes to obtain this update time: Periodic, Oversample, and Real Time Sampling. These modes are described below:

Mode	Description
Periodic	The default timing mode. To use this mode, simply place the instruction in a routine running in a periodic task. The instruction will automatically use the periodic task update rate as the update time. This mode is the easiest to implement and can be used for most applications.
Oversample	This timing mode provides complete manual control over how the instruction executes. To use this mode, configure the update rate in OversampleDT. You must then set EnableIn every OversampleDT seconds.
Real Time Sampling	This timing mode works with analog input modules to execute the instruction algorithm whenever a new analog input sample is received. To use this mode, wire the RollingTimeStamp parameter from the analog input module into the RTSTimeStamp parameter on the instruction, and wire the RealTimeSample parameter from the analog input module into the RTSTime parameter on the instruction. This mode is useful if you want the most accurate execution on instructions such as Totalizer where small errors could accumulate over time.

An example of using Periodic mode is shown in the figure below. Periodic mode can be used for the vast majority of your loops. Just make sure that the PV is sampled faster than the periodic task update rate.

Periodic mode is intended to be used by placing the block into a periodic task.

Since the timing mode is Periodic (TimingMode=0), DeltaT is automatically set to the task update time of 0.1 seconds.

For an example of where you might use Oversampling mode, consider an eddy current furnace where every 28 seconds, a new steel ingot is dropped on a conveyor and pushed forward into the furnace. An infrared camera takes a temperature reading on the ingot which is pushed out the end of the furnace. The infrared camera provides the PV (temperature) for this loop through a serial interface to the controller. In this case, a new PV is obtained about every 28 seconds, but due to the asynchronous nature of the serial port communications, there is no good way to synchronize this with a periodic task. You could use Oversampling mode to drive execution of the PIDE instruction every time a new temperature signal was received as shown below.

Turn on visibility for EnableIn pin and wire to the boolean signal indicating "new temperature received" from IR camera.

Properties - FurnaceControl

General Configuration | EUs/Limits | Cascade/Ratio | Alar

	Vis	Name	Value	Typ
I	☑	EnableIn	1	BO
	☑	PV	0.0	RE

Select Oversampling mode (TimingMode=1).

PIDE Properties - FurnaceControl

General Configuration | EUs/Limits | Cascade/F

Timing
Mode: Oversampling ▼
Oversample Δt: 0.0 s
RTS Period: 1 ms
Control action: ⊙ E = SP - PV

Furnace Control
PIDE
Enhanced PID

NewTempReceived — EnableIn   CVEU — FurnaceCurrent
IngotTemperature — PV
NewIngotRate — OversampleDT
AutotuneTag   FurnaceControlAtune

Turn on visibility for OversampleDT pin and wire to setting for the "new ingot rate." (Optionally, you could just enter a constant value.)

PIDE Properties - FurnaceControl

General Configuration | EUs/Limits | Cascade/Ratio | Ala

	Vis	Name	Value	Ty
	☐	TimingMode	1	DI
I	☑	OversampleDT	0.0	RE
	☐	RTSTime	1	DI

For an example of using Real Time Sampling mode, consider the case of a flow totalizer. In this case, a volumetric flow signal is totaled over time using a Totalizer instruction. Because the Totalizer continually adds the most recent flow sample to the running total, any small inaccuracies can build up over time. To obtain the most accurate time based samples from the analog input module, you could use Real Time Sampling mode as shown below. (Note that the Totalizer instruction internally uses double precision floating point and trapezoidal rule numerical integration to minimize any calculation errors.)
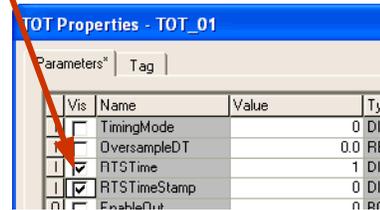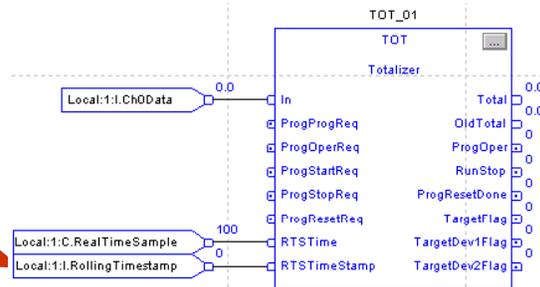
Configure the analog input module for your desired RTS rate.

Wire the RollingTimeStamp input from the analog input module to the RTSTimeStamp parameter. The totalization algorithm will now execute every time it sees the time stamp change.

Turn on visibility of RTSTime and RTSTimeStamp.

Wire the RealTimeSample value from the analog input module configuration to the RTSTime parameter.
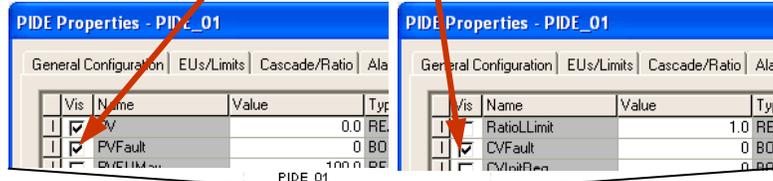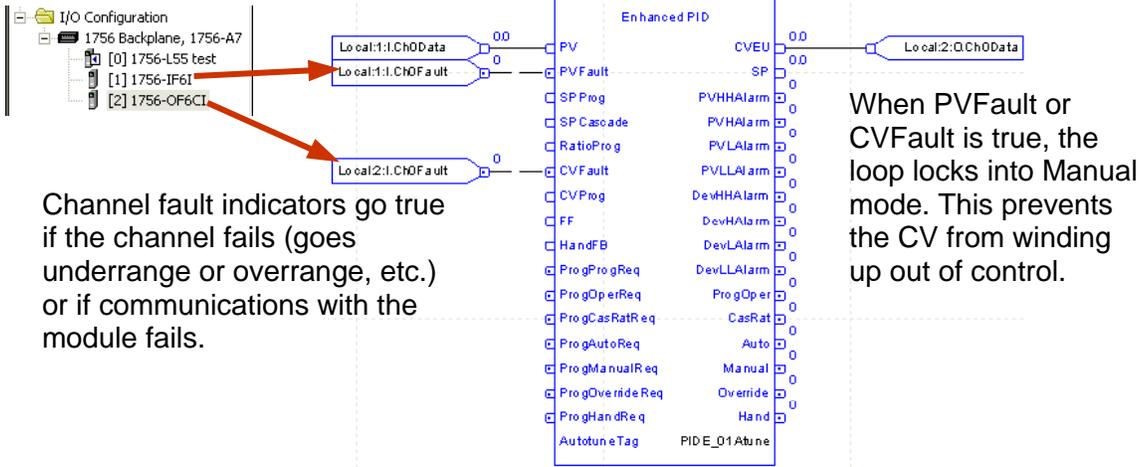

PV and CV Fault Handling

When a process variable or control variable has bad health, you don't want the PID loop continuing to try to control since it no longer has either a feedback or an actual control capability. The PIDE instruction can handle this automatically if you use 1756 I/O modules. The 1756 I/O modules all have channel fault indications for each channel. The channel fault will turn on if communications are lost with the I/O module or if faults such as underrange or overrange occur on the channel. The channel fault, therefore, is an easy single parameter to monitor the quality of the I/O channel. By wiring these channel fault indicators into the PVFault and CVFault parameters, the PIDE instruction will automatically lock itself into Manual mode any time the PV or CV has bad health.

To automatically handle PV or CV faults, first turn on visibility of the PVFault and CVFault pins.

Then just wire the analog input and analog output channel fault bits to the PVFault and CVFault pins.

Channel fault indicators go true if the channel fails (goes underrange or overrange, etc.) or if communications with the module fails.
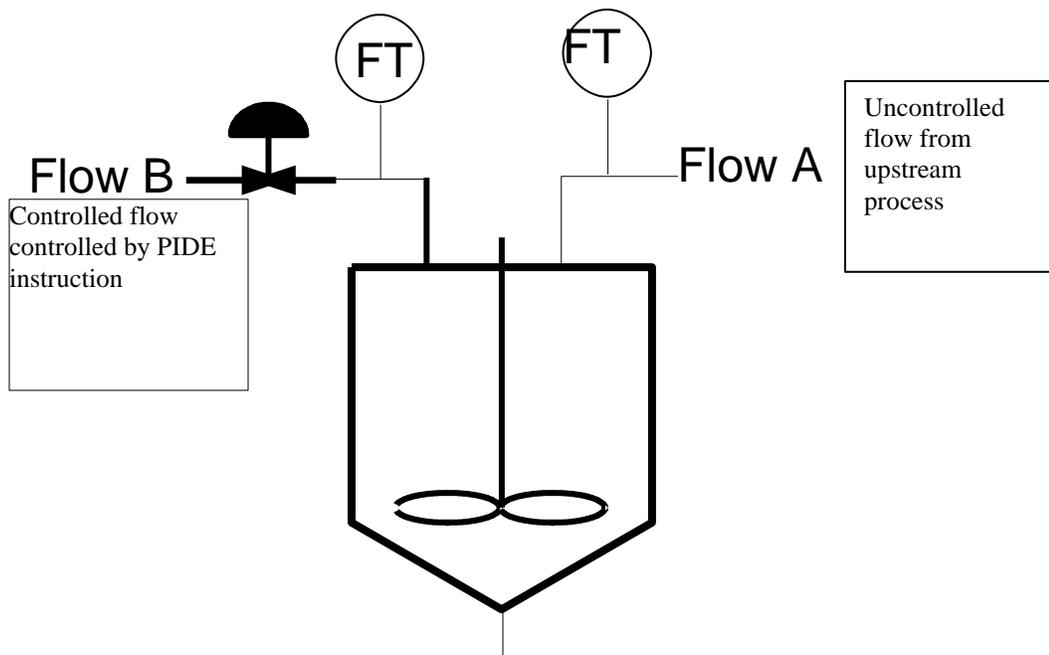
When PVFault or CVFault is true, the loop locks into Manual mode. This prevents the CV from winding up out of control.

Ratio Control

Ratio control is useful when you are trying to maintain a constant ratio of flow of one material in relationship to another. For example, assume that a continuous mixing tank receives an ingredient flow (Flow A) from an upstream source. The quantity of Flow A may vary depending on upstream processing conditions. For this reason, Flow A is often referred to as the "wild" or "uncontrolled" flow. Regardless of the amount of Flow A into the mixer, you always want to add a constant percentage, or ratio, of another ingredient (Flow B). Flow B is controlled by a PIDE instruction using Cascade/Ratio mode, where the setpoint to the PIDE instruction is determined by multiplying the Flow A signal by a ratio value.

To enable Cascade/Ratio mode, you must first set parameter AllowCasRat. This parameter is available since most loops do not need Cascade/Ratio mode, so it is disabled by default. You must also set the UseRatio parameter. This tells the PIDE instruction to multiply the SPCascade input by the Ratio value and use the result as the setpoint when in Cascade/Ratio mode. You then wire the controlled flow into the PV input and the uncontrolled flow into the SPCascade input. You can also define ratio high and low limit values to limit the ratio to valid values. An example of this implementation is shown below.

FT

FT

Flow B

Flow A

Uncontrolled flow from upstream process

Controlled flow controlled by PIDE instruction

Set AllowCasRat to allow Cascade/Ratio mode.

Set UseRatio to tell the loop to multiply SPCascade by the Ratio value.

Use the controlled flow as the PV.

Wire the uncontrolled flow into the SPCascade parameter.

Ratio limits can be used to bound the acceptable ratio values.

Ratio values can come from an operator display or the program.

PIDE Properties - RatioController

General Configuration | EUs/Limits | Cascade/Ratio | Ala

☑ Allow Cascade/Ratio mode

☑ Use Ratio

Ratio HLimit: 1.0

Ratio LLimit: 0.5

...Program.RatioController

SP PV

100.00

50.00

0.00

Operator

Ratio

Program

Operator

Cas/Rat

Auto

Manual

SP 11.25

PV 10.55

Ratio 0.75

CV

0 27.27 100 %

RatioController

PIDE

Enhanced PID

ControlledFlowB — PV | CVEU — ControlValve

UncontrolledFlowA — SPProg | SP
— SPCascade | PVHHAlarm
— RatioProg | PVHAlarm
— CVProg | PVLAlarm
— FF | PVLLAlarm
— HandFB | PVROCPosAlarm
— ProgProgReq | PVROCNegAlarm

Cascade Control

Cascade control is useful when you want to limit fluctuations in your final control element from causing upsets to your process. As an example, consider the case of a mixing tank whose temperature is controlled by the flow of steam into a heating jacket around the tank. If the steam pressure were to drop as the result of some upstream activity, the temperature of the product in the tank would start to decrease. The PID loop controlling the temperature would sense the temperature drop and eventually open the steam valve enough to bring the temperature back to setpoint. However, it would be advantageous to start opening up the steam valve **before** the product temperature was seriously affected. Cascade control provides this capability.
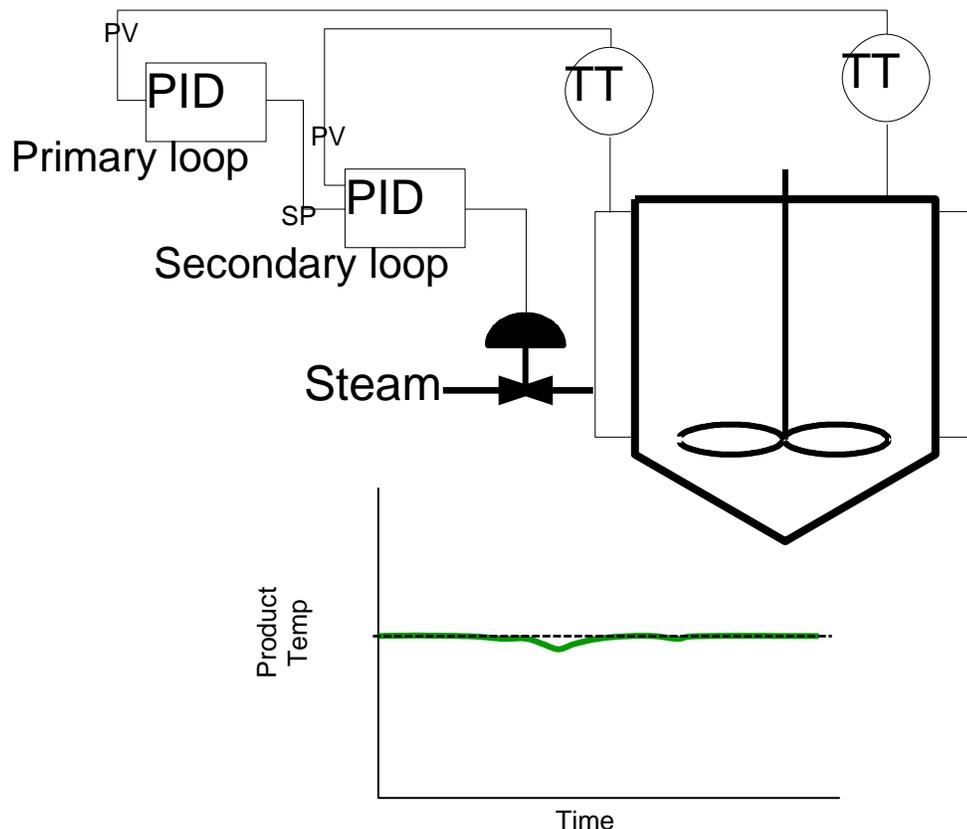
In cascade control, the loop controlling the main variable is referred to as the primary loop. This is also sometimes referred to as the master loop or outer loop. In our example, the tank temperature is controlled by the primary loop. The loop controlling fluctuations in the final control element is referred to as the secondary loop. The terms 'slave loop' or 'inner loop' are also used. In our example, you could set up a secondary loop to monitor the jacket temperature. Since the volume of the jacket is much smaller than the volume of the tank, it will respond much more quickly to changes in steam pressure. This illustrates one of the limitations of cascade control. The process response

characteristics of the secondary loop must be quicker than the process response characteristics of the primary loop. This is logical since if the secondary loop was slower, it would not be able to control disturbances before they were seen by the primary.

With a secondary loop monitoring the jacket temperature, a drop in steam pressure will now be quickly seen as a drop in the jacket temperature, and the secondary loop will start opening the steam valve before the tank temperature is seriously affected. This is illustrated in the diagram below.

- **With cascade control, a drop in steam pressure causes the *jacket* temp to drop. The secondary (inner) PID loop then responds to increased steam flow and gets the *jacket* temp back to setpoint before the *product* temp is seriously affected.**

PV

PID

Primary loop

PV

SP   PID

Secondary loop

TT   TT

Steam

Product Temp

Time

The PIDE instruction has built-in capabilities to handle cascaded

loops. First, it has a distinct mode (Cascade/Ratio) to handle cascade control. The secondary loop can either be in Cascade mode, in which case the output of the primary will provide the setpoint of the secondary, or it can be in Auto mode, in which case you can enter a temperature setpoint for the jacket directly.
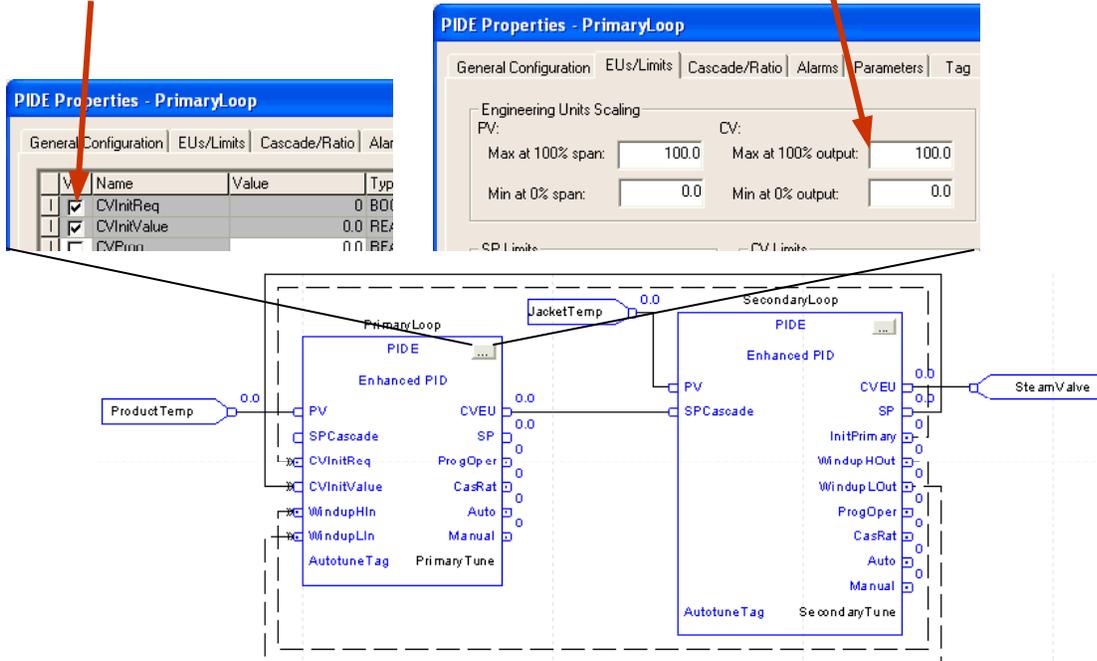
The PIDE instruction also supports initialization of the primary loop to the secondary loop's setpoint. If the secondary loop leaves Cascade mode, the primary loop needs to stop trying to control since it no longer is affecting the process. It should also set its output equal to the secondary loop's setpoint, so when the secondary is returned to Cascade mode, the primary will bumplessly start controlling. Additionally, the PIDE instruction supports windup limiting on the primary loop. When the secondary loop reaches an output or setpoint limit, you want the primary loop to stop integrating in the direction of the limit. For example, if the secondary reached a high output limit, the primary should no longer integrate in a positive direction. In our example, if the secondary loop had opened the steam valve 100%, it would make no sense for the primary to continue to ask for more steam (increase the secondary's setpoint) since the secondary cannot give any more steam.

A typical setup of a cascaded loop in RSLogix 5000 is illustrated below. First on the primary loop, you need to turn on visibility of the CVInitReq and CVInitValue pins. These will be used to setup the initialization of the primary loop when the secondary leaves Cascade mode. You should also make sure that the engineering units range of the primary's output matches the engineering units range of the secondary's setpoint since the secondary will use the primary's output as its setpoint.

Perform Common Process Loop Control Algorithms

Turn on visibility of CVInitReq and CVInitValue pins.

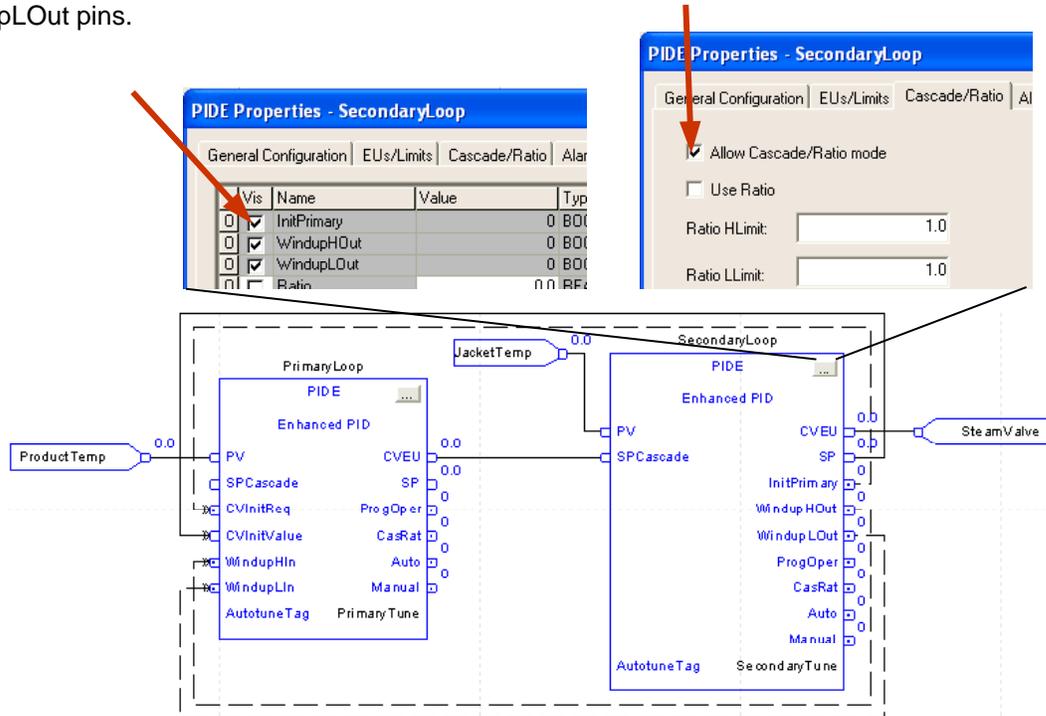Make sure CVEUMax and CVEUMin match the engineering unit range of the secondary loop's SP.

On the secondary loop, you need to turn on visibility of the InitPrimary, WindupHOut, and WindupLOut pins. These will be used to setup initialization and windup limiting on the primary. You also need to set AllowCasRat to enable the Cascade/Ratio mode just as we did for a ratio control loop.

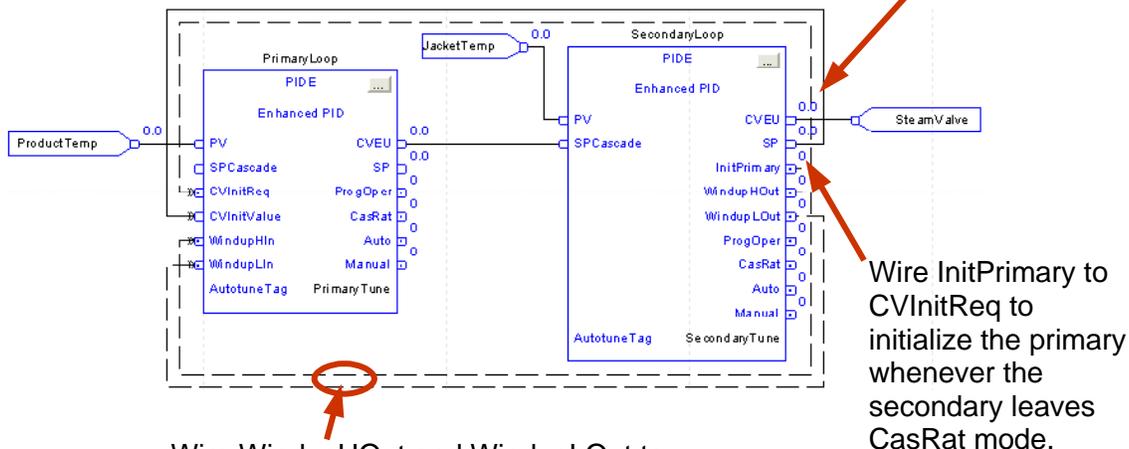Turn on visibility of InitPrimary, WindupHOut, and WindupLOut pins.

Set AllowCasRat true to enable Cascade/Ratio mode.

Finally, you should wire the InitPrimary and SP outputs of the secondary to the CVInitReq and CVInitValue inputs on the primary. When the secondary leaves Cascade mode, it will set the InitPrimary output, causing the primary loop to initialize its CVEU output to be equal to the secondary's setpoint. You should also wire the WindupHOut and WindupLOut outputs of the secondary to the WindupHIn and WindupLIn inputs on the primary. When the secondary hits an output or setpoint limit, it will set the appropriate Windup output which will cause the primary loop to stop integrating in that direction.

Wire SP to CVInitValue to initialize the primary's CVEU to the secondary's SP value whenever CVInitReq is true. This allows the primary to bumplessly line up with the secondary.

Wire InitPrimary to CVInitReq to initialize the primary whenever the secondary leaves CasRat mode.

Wire WindupHOut and WindupLOut to WindupHIn and WindupLIn. This stops the primary from integrating up or down if the secondary hits an output or setpoint limit.

One final note -- some DCS systems accomplish the primary initialization and windup limiting by wiring a single "back-calculate" wire from the secondary to the primary. This wire contains all of the initialization and windup H/L signals. However, the advantage of breaking these out as separate signals is that it allows additional flexibility for handling more advanced situations where, for example, a single primary loop might fan out to multiple secondaries.

Split-Range Time-Proportioned Loops

In certain situations, a single PIDE instruction might be used to perform two types of control depending on the output range. If we return to the example of an extrusion machine barrel zone, the temperature is controlled by pulsing resistive heaters when the PIDE output is above 50% and pulsing coolant through cooling coils when the PIDE output is below 50%. The Logix controllers support a Split-Range Time-Proportioning (SRTP) instruction for precisely these types of loops.

You also need to consider how you execute the PIDE and SRTP instructions. These types of temperature loops are usually very slow acting, so the PIDE instruction often needs to execute only every one-half to two seconds. It is important, however, that the SRTP instruction is executed much more quickly than the PIDE instruction. Since the SRTP is actually performing the pulsing of the heating and

cooling outputs, your output resolution is a function of the CycleTime of the SRTP and how often the SRTP executes.

For example, if you defined a CycleTime for the SRTP of 10 seconds, and then executed the SRTP in the same periodic task as the PIDE at once a second, your output resolution would actually only be 10%! It would be impossible to control your loops with this resolution. Therefore, what you want to do is execute the SRTP instruction in a faster, higher priority periodic task typically running every 10 or 20 milliseconds. You can then use a controller scoped tag to send the data from the PIDE output to the input of the SRTP.

For a typical loop then, your CycleTime might be 10 seconds, and if your SRTP instruction is running in a 20 millisecond periodic task, then your output resolution is 0.2% which is plenty of resolution to handle most of these types of loops. For heat/cool loops, you typically configure the SRTP instruction such that 100% PIDE output provides full heating, 50% PIDE output provides no heating or cooling, and 0% PIDE output provides full cooling. In fact, this is the default SRTP configuration. For a heat-only loop, configure the SRTP such that 100% PIDE output provides full heating, and 0% PIDE output provides no heating. Additionally, for a heat/cool loop, you will typically want to set the .CVInitValue parameter of the PIDE instruction to 50. This will cause the PIDE loop to start up with an output of 50% when the controller first goes to run mode. A typical heat/cool loop setup of the SRTP instruction is shown below.
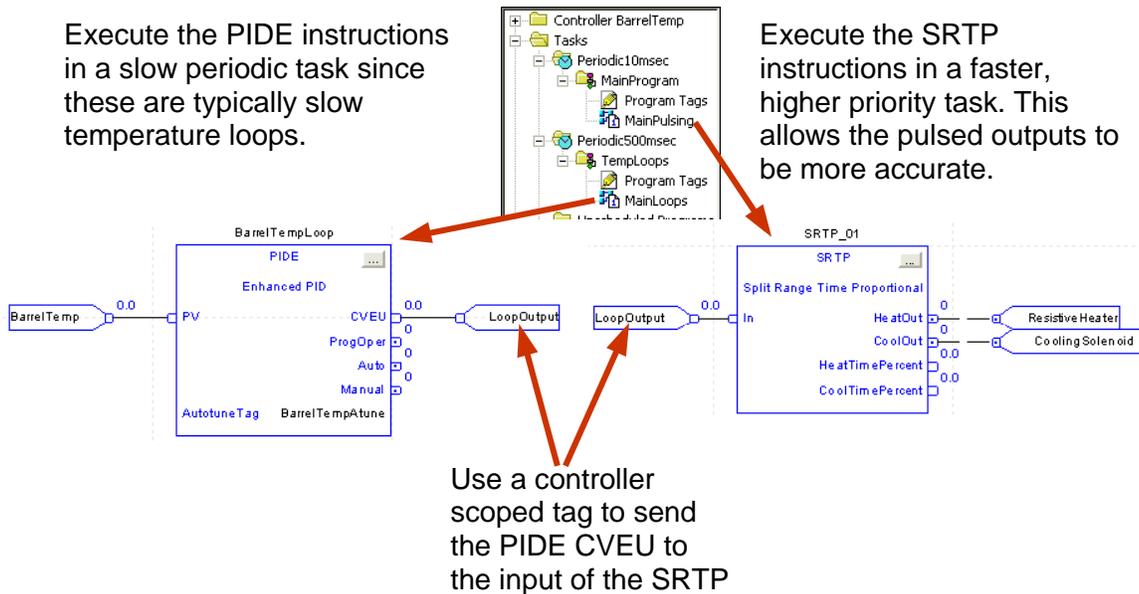
Cycle Time = 10 seconds

PIDE CVEU	SRTP % Heating	SRTP % Cooling	SRTP Heat Contact On Time	SRTP Cool Contact On Time
0%	0%	100%	0 sec	10 sec
25%	0%	50%	0 sec	5 sec
50%	0%	0%	0 sec	0 sec
75%	50%	0%	5 sec	0 sec
100%	100%	0%	10 sec	0 sec

Perform Common Process Loop Control Algorithms

The FBD logic to execute a typical heat/cool loop would then look like this:

Execute the PIDE instructions in a slow periodic task since these are typically slow temperature loops.

Execute the SRTP instructions in a faster, higher priority task. This allows the pulsed outputs to be more accurate.

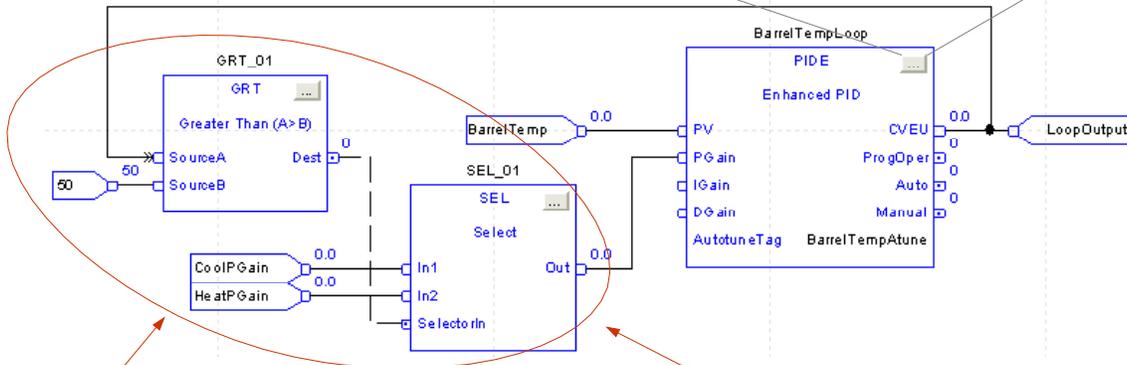Use a controller scoped tag to send the PIDE CVEU to the input of the SRTP

If you are driving an analog final control element for heating and/or cooling instead of a digital contact, you can directly use the HeatTimePercent and CoolTimePercent outputs of the SRTP instruction. They will range from 0-100% depending on the amount of heating or cooling requested by the PIDE instruction.

As mentioned earlier in this document, adaptive gains can easily be accomplished with the PIDE instruction. Often, having different gains for the heating and cooling processes can lead to better control since these are different physical processes. To accomplish adaptive gains, turn on visibility of the PGain, IGain, and DGain parameters and wire in a selection of either a set of heating gains or cooling gains as shown below.

Turn on visibility of PGain, IGain, and DGain pins. This allows the tuning constants to be programmatically changed.

PIDE Properties - BarrelTempLoop

General Configuration | EUs/Limits | Cascade/Ratio | Alar

	Vis	Name	Value	Typ
	☑	PGain	0.0	REA
	☑	IGain	0.0	REA
	☑	DGain	0.0	REA
	☐	PVEProportional	0	BOO

BarrelTempLoop

GRT_01

GRT

Greater Than (A>B)

SourceA   Dest

50

SourceB

0

BarrelTemp

SEL_01

SEL

Select

CoolPGain   In1

HeatPGain   In2

SelectorIn

Out

PIDE

Enhanced PID

PV   CVEU

PGain   ProgOper

IGain   Auto

DGain   Manual
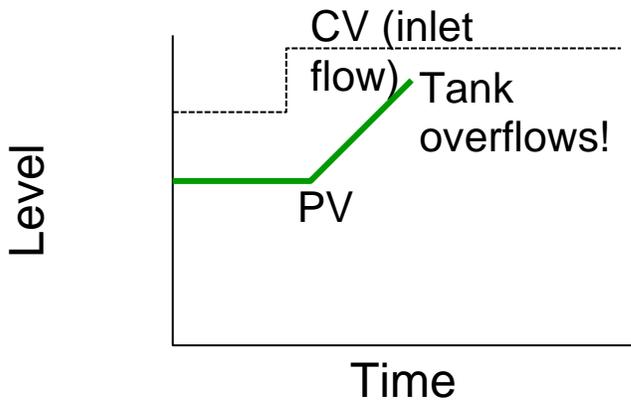
AutotuneTag   BarrelTempAtune

LoopOutput

Test if CVEU is greater than 50%. If so, select the heating gain; if not, select the cooling gain.

Repeat this selection logic for each gain.
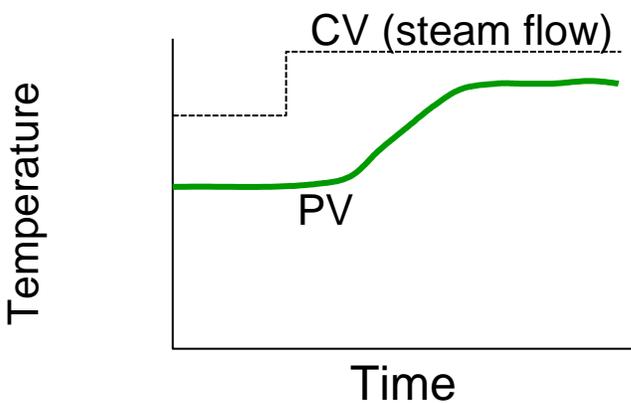
Process Simulation

For simple simulation of process loops, you can use the process instructions built right into the Logix controller. Most loops can generally be thought of as either "integrating" or "self-limiting." An example of an integrating process would be a level loop. If you consider a tank with a flow into the tank matched exactly by the flow out of the tank, it will have a steady level. If you then make a step increase in the flow into the tank, the tank will steadily fill up until it is full or overflows. This is typical of an integrating process loop; if you make a step change to the loop output, the process will steadily increase or decrease until it reaches a physical limit.

Level

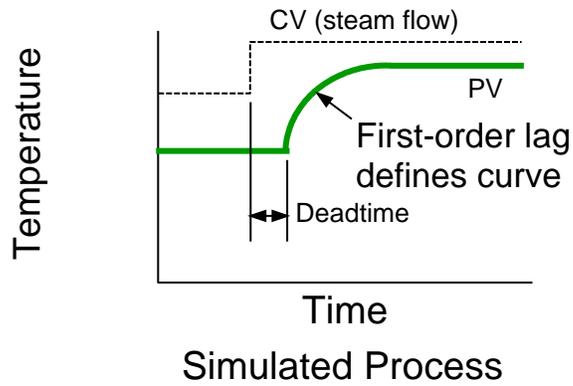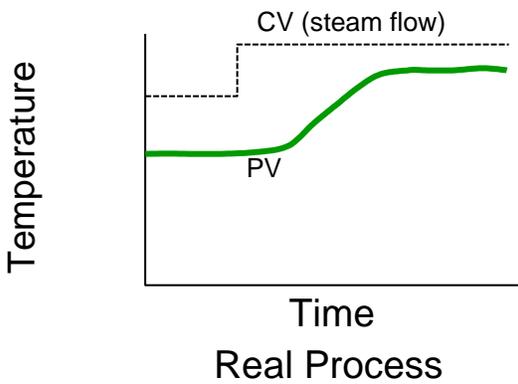CV (inlet flow) Tank overflows!

PV

Time

An integrating loop such as a level loop is simple to simulate – just perform a mass or material balance calculation on the tank.

An example of a self-limiting process loop would be a temperature loop. If you make a step increase to the loop output, the temperature might typically take a little while to start responding, and would then exponentially increase to a new steady-state value.
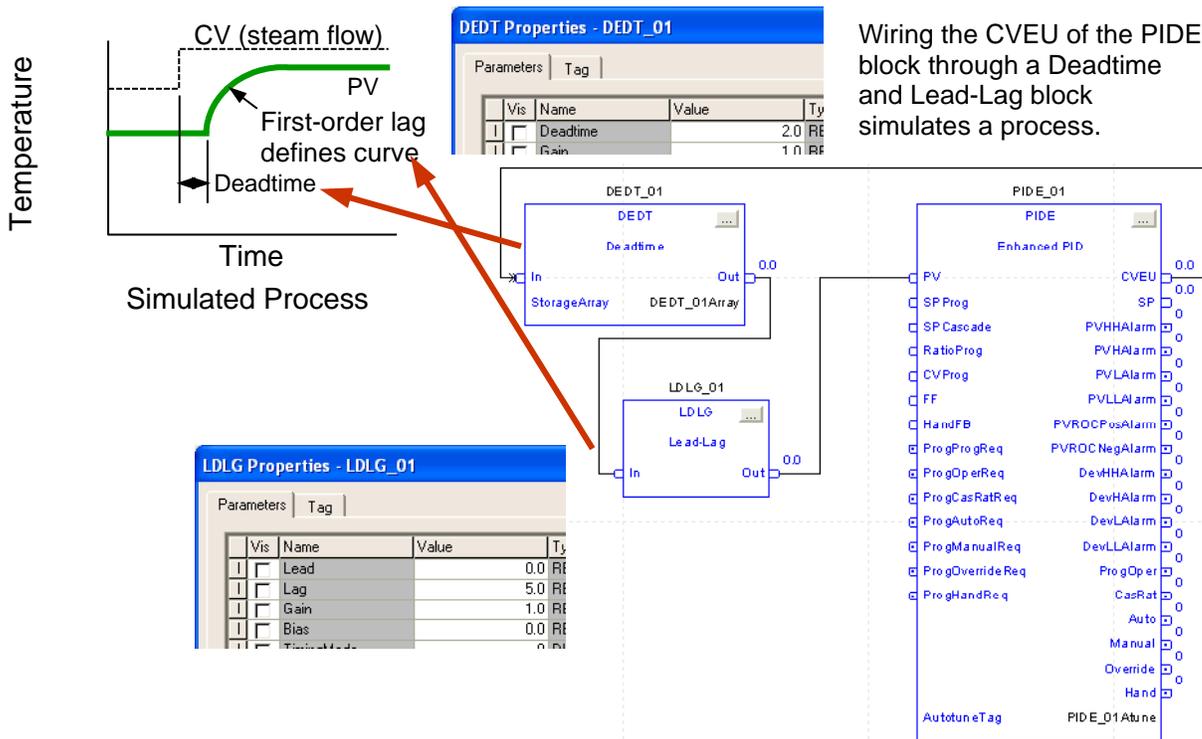
Temperature

CV (steam flow)

PV

Time

A self-limiting process loop can often be simulated by a deadtime and first order lag in series. The deadtime simulates the delay between when the output changes and the PV starts responding, and the lag simulates the exponential rise to a new steady-state value.

Temperature

CV (steam flow)

PV

Time
Real Process

Temperature

CV (steam flow)

PV

First-order lag
defines curve

Deadtime

Time
Simulated Process

The Logix controller provides Deadtime and Lead-Lag instructions which can be used for these types of simulations. The output of the PIDE instruction is wired through the Deadtime and Lead-Lag and then back into the PV input of the PIDE. The loop can then be tuned and operated with the model.

Temperature

CV (steam flow)

PV

First-order lag
defines curve

Deadtime

Time
Simulated Process

DEDT Properties - DEDT_01

Parameters | Tag

	Vis	Name	Value	Ty
I	☐	Deadtime	2.0	RE
I	☐	Gain	1.0	RE

Wiring the CVEU of the PIDE block through a Deadtime and Lead-Lag block simulates a process.

LDLG Properties - LDLG_01

Parameters | Tag

	Vis	Name	Value	Ty
I	☐	Lead	0.0	RE
I	☐	Lag	5.0	RE
I	☐	Gain	1.0	RE
I	☐	Bias	0.0	RE
I	☐	TimingMode	0	DI

DEDT_01

DEDT

Deadtime

In   Out   0.0

StorageArray   DEDT_01Array

LDLG_01

LDLG

Lead-Lag

In   Out   0.0

PIDE_01

PIDE

Enhanced PID

		0.0
PV	CVEU	0.0
SPProg	SP	0.0
SPCascade	PVHHAlarm	0
RatioProg	PVHAlarm	0
CVProg	PVLAlarm	0
FF	PVLLAlarm	0
HandFB	PVROCPosAlarm	0
ProgProgReq	PVROCNegAlarm	0
ProgOperReq	DevHHAlarm	0
ProgCasRatReq	DevHAlarm	0
ProgAutoReq	DevLAlarm	0
ProgManualReq	DevLLAlarm	0
ProgOverrideReq	ProgOper	0
ProgHandReq	CasRat	0
	Auto	0
	Manual	0
	Override	0
	Hand	0
AutotuneTag	PIDE_01Atune	

By choosing different values of deadtime and lag, you can simulate different types of process loops. For example, a slow temperature loop might have a deadtime of a few minutes and a lag time constant of several minutes, while a fast flow loop might have a deadtime of a couple seconds and a lag time constant of a few seconds.
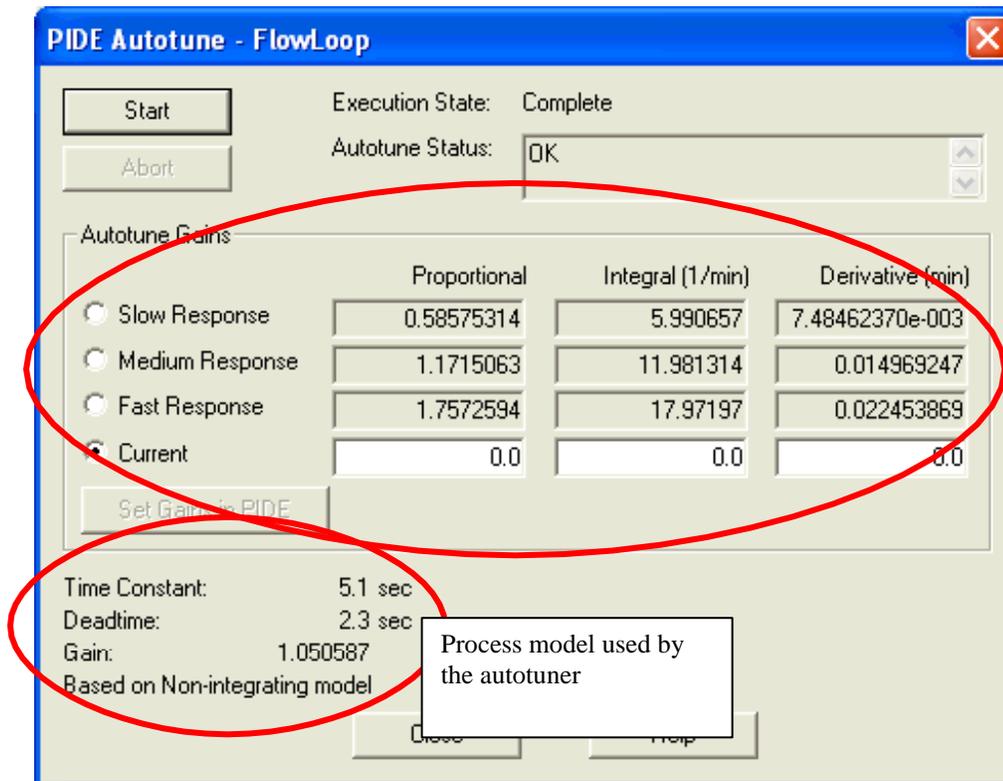
You can also use the Gain parameter on the Deadtime or Lead-Lag instruction to simulate a process gain. For example, if a 10% change in loop output would typically cause a 20% change in PV, you could use a Gain of 2 to simulate this behavior. Similarly, if your loop has an ambient condition whereby a loop output of 0% would cause the process to settle at some non-zero value, you can enter this value as a Bias. For example, a temperature loop might settle at room temperature if the loop output was 0%. Finally, you might sometimes also want to use a Scale (SCL) block to scale the output of the PIDE instruction into a PV value with a different range.

Autotuning

The PIDE instruction has a built-in autotuner which you can use to obtain suggested tuning constants for your process loop. Because the autotuner is built into the PIDE instruction, you can tune your loops within RSLogix 5000 or from any operator interface. The PIDE autotuner is an open loop autotuner, meaning that the loop must be in manual. The autotuner will step the output by an amount you configure, watch the response of the PV, and then give you sets of suggested proportional, integral, and derivative gain values for a fast, medium, or slow response. As shown below, in addition to the suggested tuning constants, the autotuner also returns the process model which was used to estimate the tuning constants. By comparing this process model to the actual process, you can get an idea of the appropriateness of the suggested gains.

Tuning constants suggested by the autotuner

PIDE Autotune - FlowLoop

	Execution State:	Complete
Start	Autotune Status:	OK
Abort		

Autotune Gains

	Proportional	Integral (1/min)	Derivative (min)
Slow Response	0.58575314	5.990657	7.48462370e-003
Medium Response	1.1715063	11.981314	0.014969247
Fast Response	1.7572594	17.97197	0.022453869
Current	0.0	0.0	0.0

Set Gains in PIDE

Time Constant: 5.1 sec
Deadtime: 2.3 sec
Gain: 1.050587
Based on Non-integrating model

Process model used by the autotuner

If more autotuning capability is desired, the PIDE instruction also supports the RSTune and RSLoopOptimizer packages. These PC-based autotuners support closed loop tuning and also, particularly in the case of RSLoopOptimizer, provide a wealth of diagnostic information regarding your process loops.

Summary

For more reference information on the Enhanced PID instruction and the rest of the process control instruction set, you can refer to the Logix5000 Controllers Process Control and Drives Instructions Reference Manual, publication 1756-RM006. This manual gives a detailed description of the operation of each of the built-in process instructions.

The Enhanced PID instruction goes beyond the traditional realm of PLC-based loop control by providing a host of advanced features, allowing you to easily set up more advanced loop algorithms without the onerous ladder programming required by traditional systems in the past. However, the PIDE instruction is only one piece of a Logix-based process solution. Other features such as the entire process control instruction set, full-featured Function Block Diagramming, Sequential Function Chart, and Structured Text editors, ControlLogix redundancy, a huge selection of I/O options, including HART and FOUNDATION Fieldbus, and integration with our RSView operator interface solutions, allow the Logix controllers to provide a solution as adept at performing process control as they are at sequential, motion, or drives control. This provides the opportunity to drastically decrease your engineering and maintenance costs by leveraging a common, scaleable platform across your entire facility. Whether you are controlling continuous or batch process applications, high-speed packaging machines, or coordinated drive systems, Logix now has the capabilities to handle all these applications.