Temperature Chamber Documentation

# Introduction

This example demonstrates the implementation of a simple temperature chamber controller application using the Tag Bus Data (TBD) Framework. The goal of this example is to provide a baseline understanding of how the framework works, of the benefits that it provides, and of the workflow required to build your own applications. The example includes two implementations. The first makes use of a simulated system with a simple model of a temperature controller. The second implementation uses real-world I/O in combination with a model and requires both a cRIO and a temperature chamber to make full use of its features.

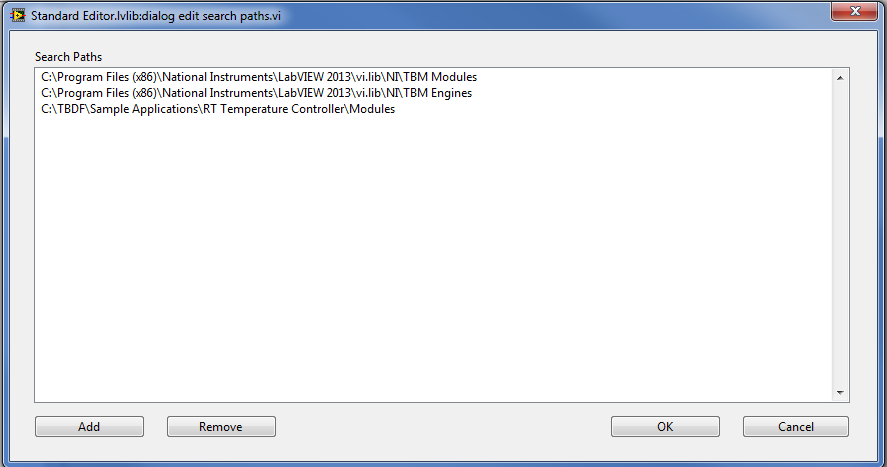
# Simulated System Example

## Configuration Editor

TBD allows users to specify a large portion of their application’s behavior through a configuration file. As a result, viewing the system configuration file is often the best place to start for understanding a TBD implementation. Open up the standard configuration editor for TBD by navigating in LabVIEW to **Tools>>TBDF>>Launch Standard Configuration Editor…**

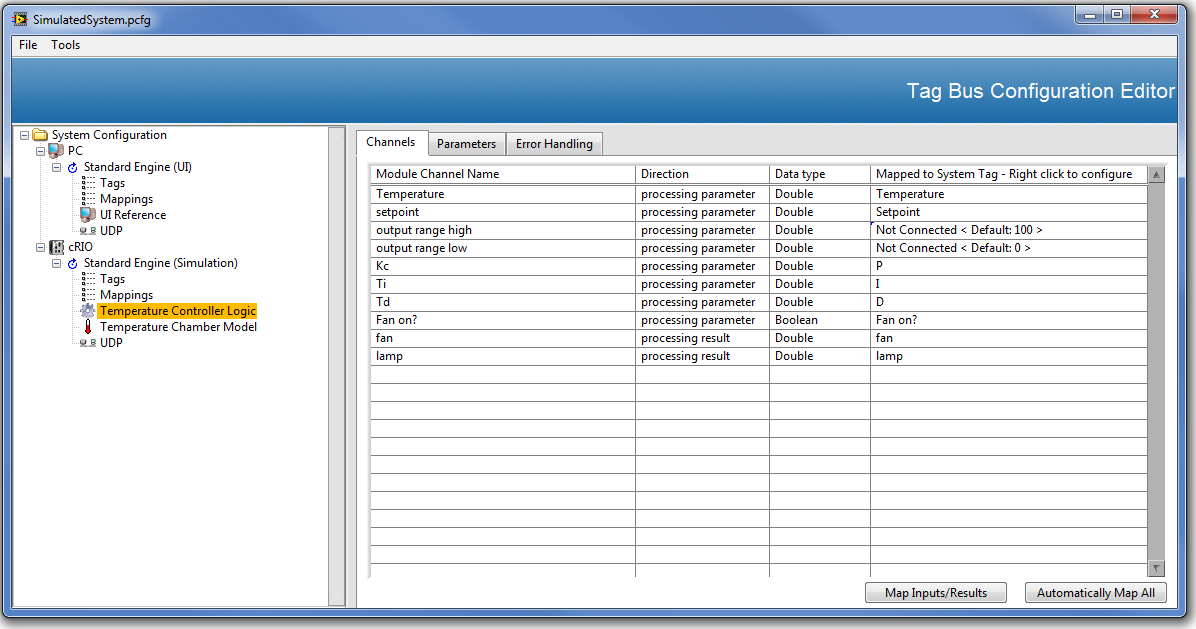
This is the default editor for the framework and is used to view and modify system configurations. Before loading a configuration, first ensure that the editor can find all of the Tag Bus Module (TBM) plug-ins that are a dependency of that configuration. To do this, navigate within the editor to **Tools>>Edit Plugin Search Paths**.

Here you will need to add a search path to the TBM plugins for this example at …<file path>. Also confirm that the standard vi.lib file paths are specified as shown below for the version of LabVIEW that you are using.



<UPDATE WITH CORRECT INSTALL LOCATION>

Now hit ‘OK’. The configuration editor will now scan these directories for any TBM plugins and load them into memory. Once the busy cursor disappears, open up the ‘SimulatedSystem.pcfg’ file (stored in the same directory as the example project) using **File>>Open**.

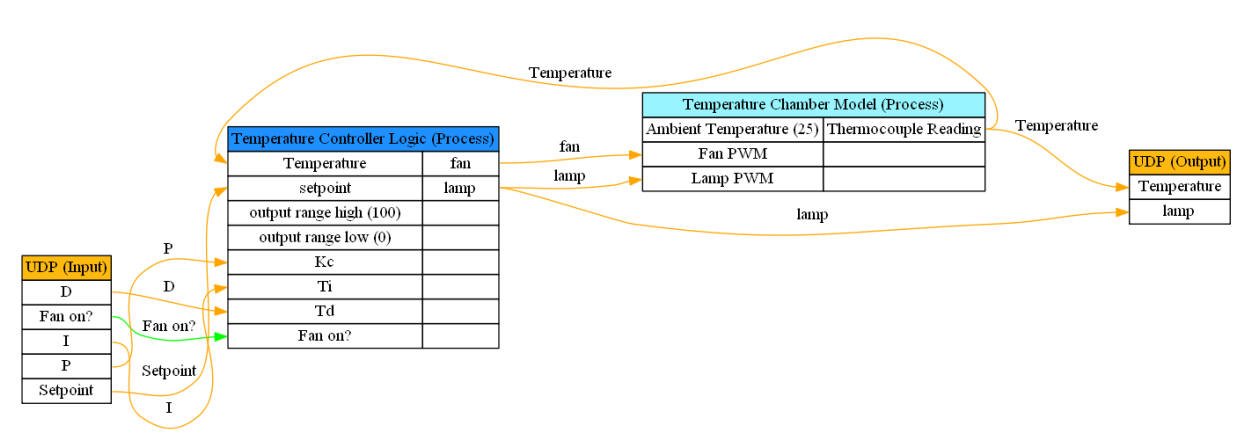


You should now see a tree control populated on the left side of the editor with a hierarchal set of items. Clicking on an item in the tree control populates the view on the right which can be used to edit that item’s configuration. Feel free to explore the editor by clicking various nodes within the tree control. The top-level node is the System. It has properties for the configuration version and description. Each System is comprised of one or more Targets that have an IP address, operating system and other properties. Each Target configuration is then comprised of one or more Engines.

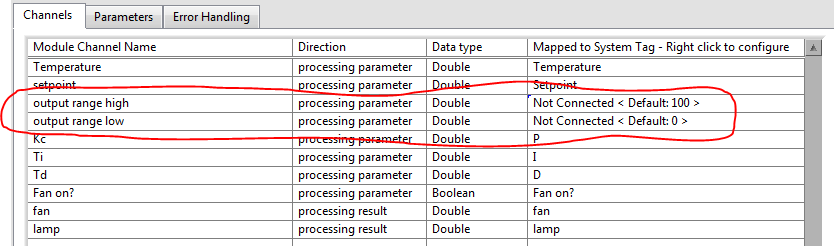
An Engine is essentially a background process with a timing source and a collection of named locally-scoped current value data (Tags). Each Engine can be configured to execute one or more Tag Bus Modules (TBMs). Mappings are used to specify the exchange of data within a TBM (called a Channel) with the Tag data in the engine. Two TBMs can share data with each other by mapping their input and output channels to the same Tag alias.

For this particular system configuration, you will see that the cRIO has an Engine that is configured to execute both a model (which simulates the temperature chamber and its I/O) and the temperature controller logic (in this case just simple PID). Inspect the ‘Tags’, ‘Mappings’, ‘Temperature Controller Logic’, and ‘Temperature Chamber Model’ under the ‘Standard Engine (Simulation)’ node to see how the Channels of the model and controller are connected through Tags.

The diagram below provides a different visual representation of TBMs, Channels, and Tags. In the diagram, each colored row represents a TBM method. The entries under the TBM method represent the Channels of that TBM. The free floating labels represent Tags, and the connection of a Channel to a Tag to another Channel is represented as a wire.



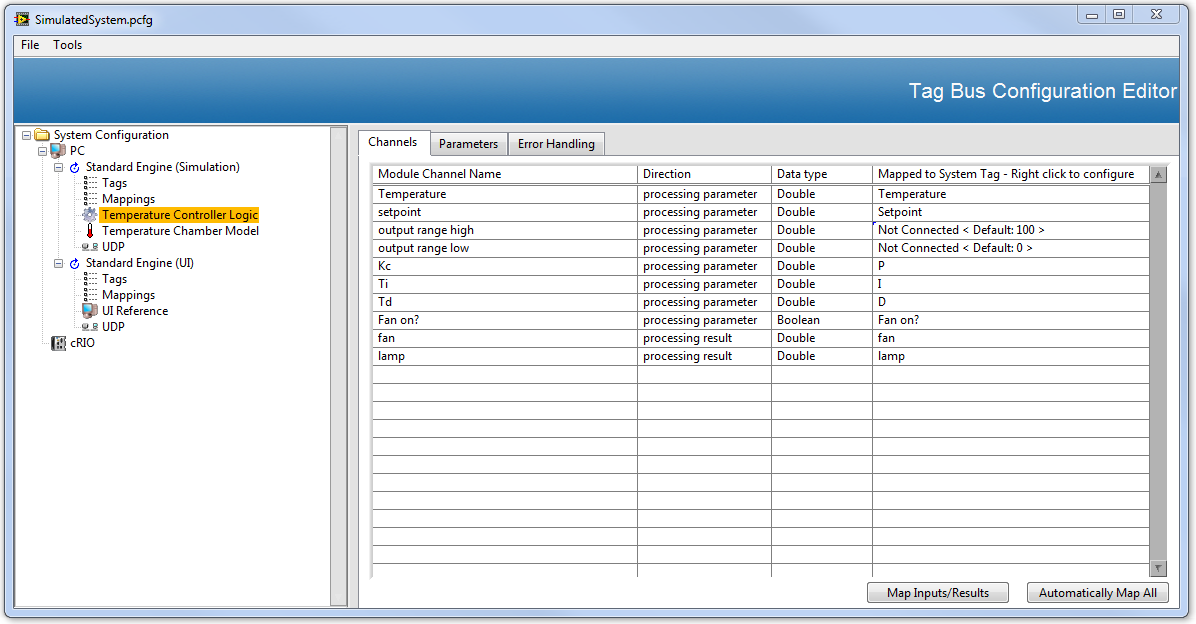
Also notice that both the ‘Temperature Controller Logic’ and ‘Temperature Chamber Model’ have at least one Channel that isn’t mapped to a Tag in the Engine. For any Channel that is unmapped, the default value of that Channel can be specified instead.



In addition to calling TBMs and exchanging data between them, the engine is also responsible for determining the execution rate of the system as well as the error handling configuration for each module. Click on the ‘Standard Engine (Simulation)’ node in the tree to see how these are configured for this system.

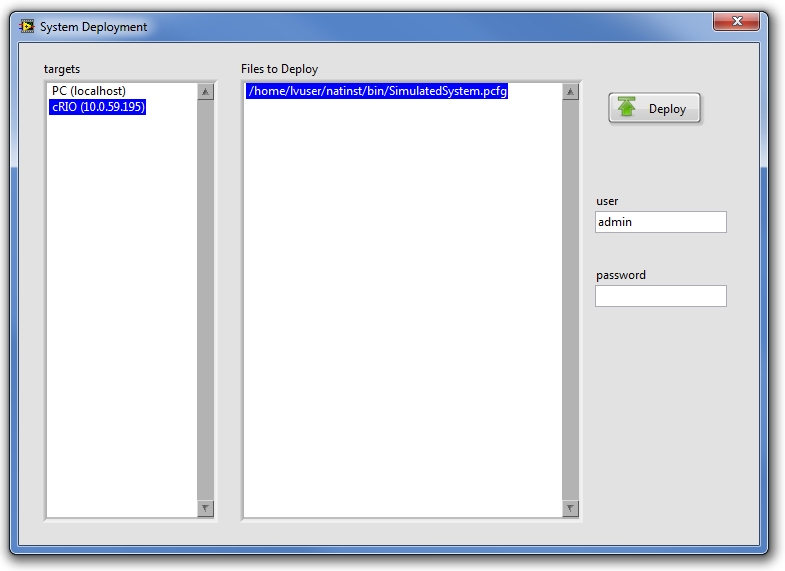
The PC in this configuration is serving as a UI. The UI provides the setpoint for the temperature controller as well as the command signal to turn on the fan disturbance. Status information regarding the operation of the controller is then returned and presented on this UI. Data is transferred from one engine to the other using two instances of a UDP TBM. This module has an instance on the PC, and a second instance on the cRIO, and the two are paired together.

If you don’t have access to a cRIO, you can easily modify this configuration so that the Engine on the cRIO runs on the PC instead. To do this, click and drag on the ‘Standard Engine (Simulation)’ node under the engine and drag it up to the PC.



For PC only execution, make sure that the ‘send to address’ field under ‘Module Settings’ for both UDP Modules are set to ‘localhost’. For cRIO users, make sure that the ‘send to address’ for the PC UDP module has the IP Address of the cRIO and vice versa for the cRIO UDP module.

Finally if you are planning to execute code on the cRIO, specify the IP Address of your cRIO in the ‘cRIO’ configuration window. Save your configuration file to apply any changes. Then navigate to **Tools>>Deploy** Tool to transfer your configuration file to the cRIO target.

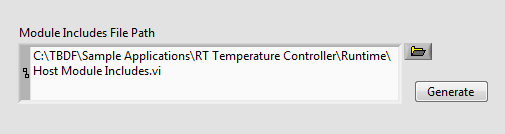


## ‘Host Main.vi’

Now that we’ve explored the system configuration for this example, let’s take a look at the LabVIEW code that will execute it. If you haven’t already, open up ‘Temperature Controller Example.lvproj’. Then open up ‘Host Main.vi’ from the project.

This is a simple example that opens up a configuration file, loads the engines and modules that the configuration contains, and then runs them until told to stop or an error occurs. The majority of this code is generic to any system configuration with two exceptions. The first is the ‘User Interface.vi’ which has UI elements specific to the example. Notice that the Block Diagram of this VI is essentially empty. This is because the ‘UI Reference’ TBM is able to write and read to controls and indicators directly as specified in the configuration. The ‘UI Reference’ TBM just needs the name of the VI and the specific controls and indicators to interact with.

The other application specific code resides in the ‘Host Module Includes.vi’. The purpose of this VI is to load into memory the Engines and TBMs required by the configuration file. This VI should be updated anytime a TBM is deleted or a new TBM is added to the configuration. A Module Includes VI can be kept synchronized automatically by adding the path to the VI on the Module Includes page of the Target and clicking ‘Generate’.



<UPDATE WITH PROPER PATH>

The framework was also designed so that TBMs could be built and deployed as plugins on disk which get loaded into memory dynamically at runtime instead of being statically included. When taking this approach the ‘Host Module Includes.vi’ would no longer be necessary, but you would need to build a source distribution for each TBM and keep the build up to date as the TBM code changes.

To recap, the ‘Host Main.vi’ will load and execute a system configuration created by the editor. It also has two clearly defined locations for placing application specific code related to the framework, one of which can be kept up to date automatically. It’s worth pointing out that it’s also possible to implement additional functionality alongside the framework as part the main VI. TBD doesn’t require that its used for every aspect of an application and can instead be applied for the tasks that make the most sense.

Now go ahead and run the ‘Host Main.vi’. You should see the following User Interface appear which will allow you to specify the temperature setpoint, tune the PID gains, turn the disturbance on and off, and monitor the resulting temperature and lamp intensity. If you are running this example purely on the PC you should see data appear, otherwise it will appear as below until the ‘cRIO Main.vi’ begins running as well.

## 

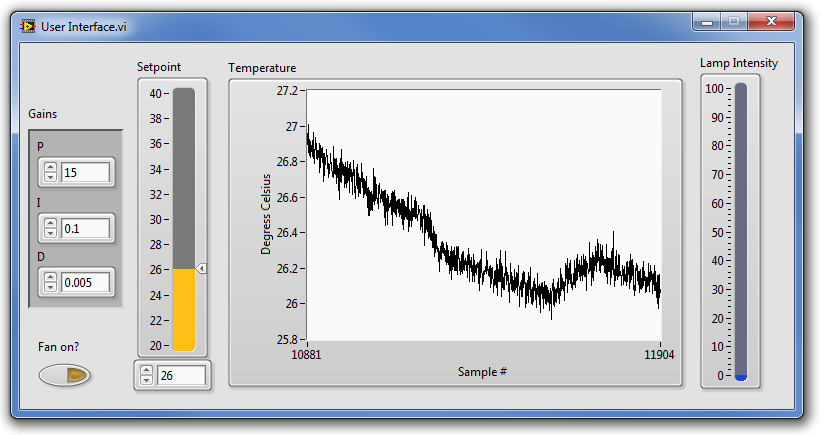
## ‘cRIO Main.vi’

Now let’s take a look at the cRIO application. Open up ‘cRIO Main.vi’ in the project. This application is very similar to the ‘Host Main.vi’ except that the Engine API functions are contained within a background service that receives commands through a queue. This service is useful because it allows system configurations to be loaded and unloaded repeatedly without requiring a reboot to the controller. It also allows a developer to pipe commands to start and stop the engine over the network instead of generating them locally.

Like the ‘Host Main.vi’, the ‘cRIO Main.vi’ has two places where application specific code resides. There is a ‘cRIO Module Includes.vi’ which serves the same purpose as the ‘Host Module Includes.vi’. There is also a debugging loop with a collection of application specific Current Value Table (CVT) tags.

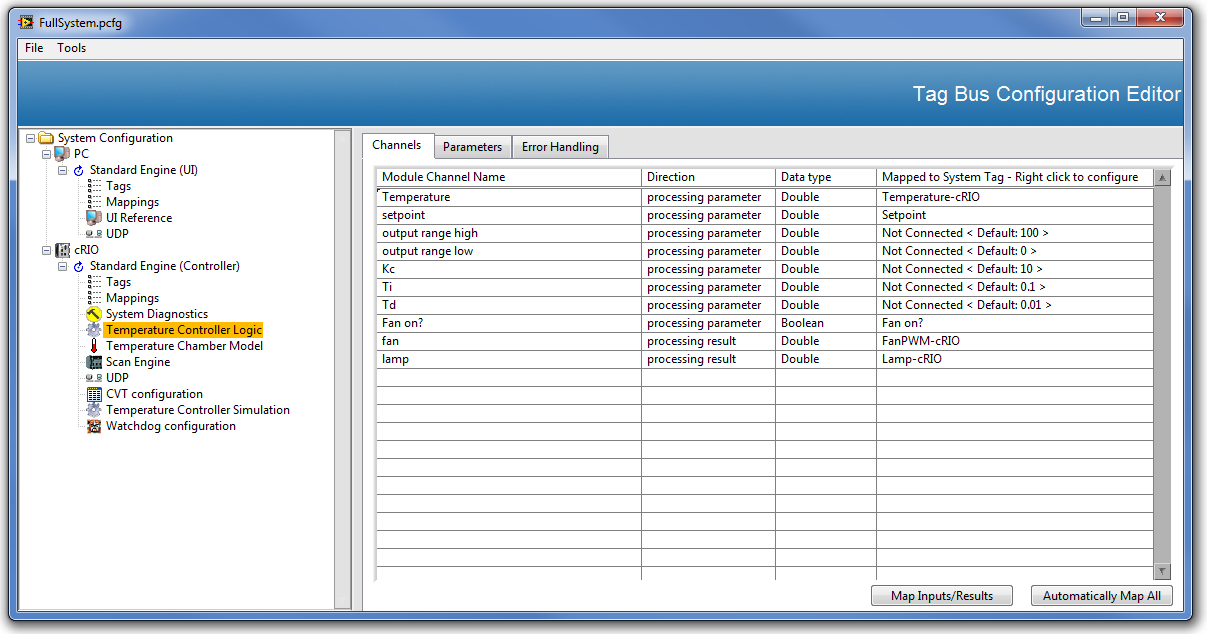
One great way to use TBD is as a data engine. You can configure I/O through the editor, map that data to CVT tags, and then access that data throughout your application using the CVT API. The ‘FullSystem.pcfg’ file, which we will examine next, provides an example of how to do this. For the ‘SimulatedSystem.pcfg’ these CVT tags are not used.

If you are using a cRIO with this example, and the configuration file is deployed to the target, go ahead and run ‘cRIO Main.vi’. You should now see data appear on the ‘Host Main.vi’. (If you don’t, make sure that your IP settings for the UDP module are specified properly in the system configuration and that your firewall isn’t blocking the UDP communication.)



# Full System Example

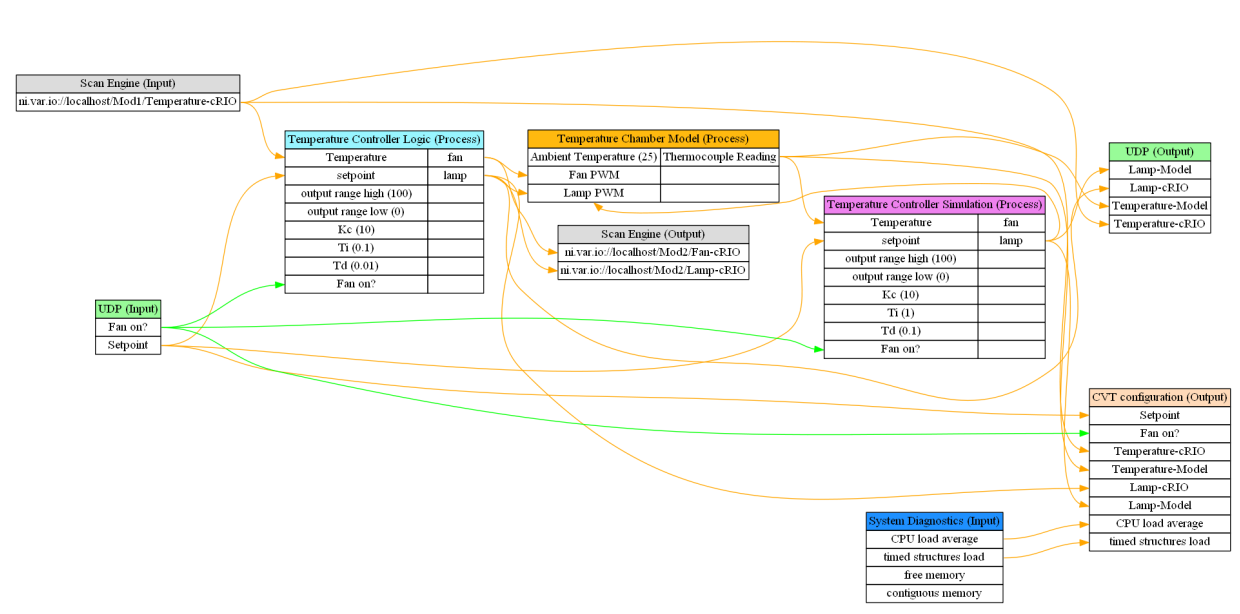
Whether or not you have access to the hardware necessary to run the full example, it is still worthy of study. Go back to the Standard Configuration Editor and this time open up the ‘FullSystme.pcfg’.



The first thing to notice is that the cRIO target has been configured to run additional modules. It now includes the Scan Engine TBM to interact with real world IO and an additional controller for the real temperature chamber. This means that the cRIO is configured to simultaneously control both a model and the real world.

There are also some TBMs added for utility. The first is System Diagnostics which returns CPU and memory usage of the system. The next is the Watchdog module which will automatically reboot the controller if it doesn’t run at least once during the timeout period. The last additional TBM is the CVT which is used to expose the data from certain Engine Tags for access by any code running on that target. Reopen the ‘cRIO Main.vi’ and see how its CVT tags are configured in the editor by the CVT configuration.

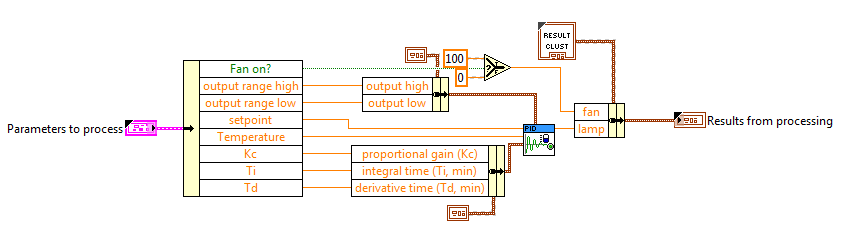
The diagram below provides visual representation of the TBMs, Channels, and Tags for this configuration. The diagram uses the same conventions as the one above, but omits the Tag labels for the wires.



# Fixed-Channel TBMs

Many TBMs can be reused for any project. This example makes use of the ‘CVT’, ‘UDP’, and ‘UI’ TBMs which can and have been used in a variety of applications. However, the control logic running in an application is often specific to that application. The Fixed-Channel TBM was created for this use-case and was utilized to build the ‘Temperature Controller Logic’, ‘Temperature Chamber Model’, ‘Diagnostics’, and ‘Watchdog’ TBMs. These TBMs are all similar in that they have a fixed number of inputs and outputs.

Now open up <vi.lib> and open the Block Diagram.

 As you can see, this VI uses a cluster ‘Parameters to process’ to define its inputs, and a different cluster ‘Results from processing’ to define its outputs. The data in these clusters corresponds to Processing Parameters and Processing Results respectively. Changes to these clusters require rerunning the Fixed Channel Module Script from LabVIEW **Tools>>TBDF>>Fixed Channel Module Script…**  The script uses the information in these clusters to generate the remaining TBM code including the editor UI. In the simplest case, creating a TBM to execute code inline with the engine can be as simple as using the Fixed Channel Sample Project and placing your logic within a single VI.

# Building Your Own Application

We have now gone over the configuration editor, the Host Main, the cRIO Main, and the Fixed Channel TBM and explored how they can be put together to implement a simple application. When using these tools to build your own applications, you’ll probably want to start by creating your own ‘Host Main.vi’ or ‘cRIO Main.vi’ using either the XXXX or XXXX sample project respectively.

Next you’d likely want to decide whether you want to create your own Fixed Channel TBM and/or use the CVT to interact with Engine Tags.

The framework also makes it possible to build your own reusable I/O or utility TBMs, to customize the configuration editor, or even to create your own TBM Execution Engine.

How do we want to wrap this up?