Distributed Control and Automation Framework (DCAF) Hands-On Session

This hands-on covers the basics of implementing an application in the Distributed Control and Automation Framework, including using an existing module and developing a new control module. It doesn’t cover development of a new generic I/O or processing module. For this hands on, the framework downloads, and additional documentation, visit ni.com/dcaf

## Set-up

To install DCAF in LabVIEW open VI Package Manager (VIPM), search for DCAF, and install in the corresponding LabVIEW version. This package can be installed to any LabVIEW version from 2014 to present. A CompactRIO controller is not required.

**Introduction**

Most control applications have similar challenges and needs. By working with different large control applications, we managed to identify most of this common challenges and needs and created DCAF to provide a standard framework to develop control applications.

Most of these challenges are related to having different processes running in parallel that need to share data without falling into race conditions. DCAF provides the capability of creating synchronized engines to run standard and custom modules and defining the mapping of data between them through a simple interface known as the Configuration Editor.

Before we start with the Exercises we need to understand some basic terminology of how a DCAF system is structured.

**System:** Your System will consist of one or more targets containing the one or more ***DCAF Engines***.

**Target:** A ***Target*** will represent the physical device that will run one or more ***Engines***. A ***Target*** could be a PC or a CRIO.

**Engine:** The Engine will be in charge of executing ***Modules*** in a synchronous way and transfer data between them through the ***Tag Bus***.

**Module:** Piece of code with a specific functionality that will be executed within an Engine. Some standard Modules are installed with DCAF, but you can create your own modules.

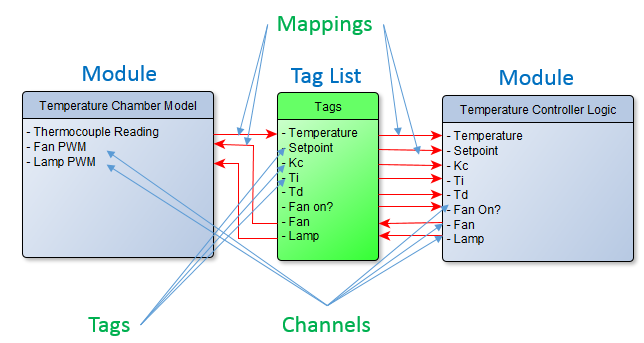
Once we have defined the terminology to understand the hierarchy of a DCAF system, we need to understand how the data flows through a DCAF system. Data can be passed between ***Modules***, ***Engines*** and even ***Targets***. Here is some more terminology related to dataflow in DCAF.

**Channels:** Parameters that allow access to and from a ***Module***. Channels can be Inputs, Outputs, Processing Parameters and Processing Results – note that the direction is taken to be from the engine’s point of view

**Tags:** Scalar variables saved in a single repository (***Tag Bus***) that can be accessed by any ***Module*** within an ***Engine***. A ***Tag*** can be defined as a connecting point between ***Channels*** from different ***Modules***.

**Mappings:** Mappings are the connections between ***Tags*** and ***Channels***. If you want a specific ***Channel*** to write or read a value on a specific ***Tag*** you will have to map them.

Take the following example to clarify the previous terminology. Let’s say a Module called ***Temperature Chamber Model*** has an Input Channel called ***Thermocouple Reading***– the module implements reading from a thermocouple and puts the value into the **Thermocouple Reading** channel. This ***Thermocouple Reading Channel*** is mapped to a Tag called ***Temperature*** – the engine will then take the value that the module places onto the channel and put it on the tag. Then the ***Temperature*** Tag’s value is passed by the engine to ***Temperature***, an Output Channel that belongs to a module called ***Temperature Controller Logic***.

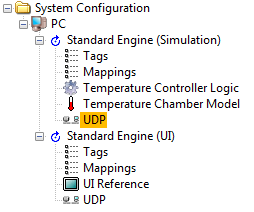


# Exercise 1: Configuring a Simple Temperature Control Application

This exercise demonstrates the implementation of a simple temperature chamber controller application. It makes use of a model of the chamber to simulate its I/O and allows users to define the setpoint and PID gains of the control algorithm through a simple user interface.

During the exercise you will learn to identify how inputs and outputs from different modules are mapped within DCAF to provide communication between modules. You will also learn how to create a UI and map it to data within the framework.

Our ***Simulated Temperature Controller*** will consist of 2 ***DCAF Engines***: the ***UI*** and the ***Temperature Controller Simulation***.



*Figure 1.1*

In the hierarchy shown above you can find some of the components defined in the previous section. In each of these ***Engines*** you will find ***Modules***. Some of these ***Modules*** are standard and some of them were created specifically for this Hands On.

**Standard Modules**

**UDP:** This module exists in both components. It is designed to share tags between Engines by mapping each tag as an Engine Input or Output. All tags that are intended to be shared between engines need to be defined in the Tags Pane of each engine with the same names.

**UI Reference:** This module takes a pre-existing front panel and maps its controls and indicators to DCAF tags to permit direct user interaction with the framework.

**Custom Modules**

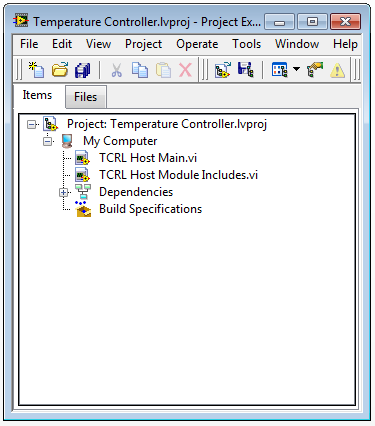
**Temperature Controller Logic:** This is a custom DCAF Module designed to provide the control logic for the temperature chamber. If the Simulation Engine is moved to a cRIO Target and the Temperature Chamber Model is replaced with real IO, this module could remain the same.

**Temperature Chamber Model:** This module provides a simulated model of a Temperature Chamber. This module could be replaced or overwritten to eventually provide IO from a real Temperature Chamber.

**Part 1: Project Creation and UI**

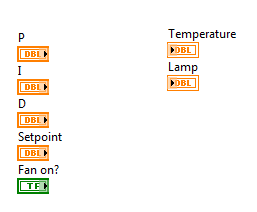
During this first part of the exercise you will be create a DCAF project from scratch using a template and learn how to add a User Interface to your DCAF project.

1. In LabVIEW go to ***File >> Create Project*..** and select DCAF. From the displayed list select ***Basic Execution Template*** and press the ***Next*** button.
2. Name the project ***Temperature Controller*** and select [***\\Desktop\DCAF Hands On\Exercises***](file:///\\Desktop\DCAF%20Hands%20On\Temperature%20Controller\Exercise%201)***\Temperature Controller\Runtime\Temperature Controller*** as the Project Root. Type ***TCRL*** as the ***File Name Prefix***.
3. Verify your project window matches Figure 1.2.



*Figure 1.2*

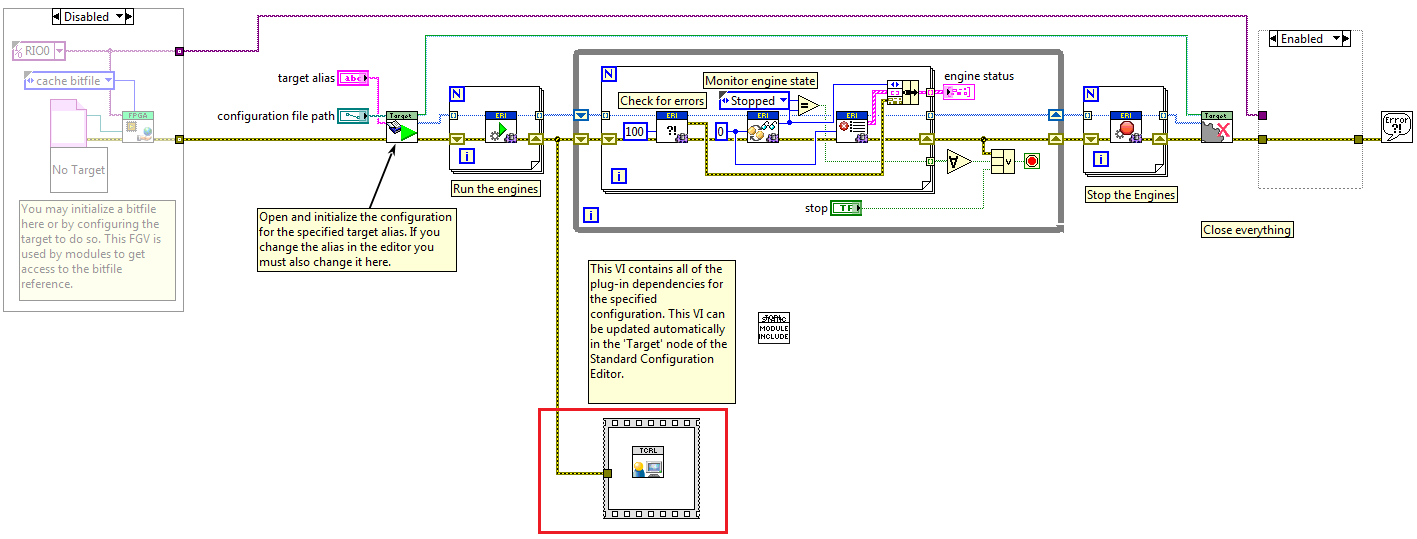
1. Add to the project a Configuration File with the Engines for the Temperature Controller partially configured and mapped (in future exercises you will make a Configuration File). Under ***My Computer*** add ***SimulatedSystem.pcfg*** located at ***\\Desktop\DCAF Hands On\Exercises\ Temperature Controller***.
2. To speed up the exercise, a UI has already been created. Under ***My Computer***, add ***TCRL User Interface.vi*** to the project located at ***\\Temperature Controller\Runtime*.**
3. Look at the names of the Labels in the Block Diagram. This is important to correctly map the tags to the UI. Controls and Indicators will be directly updated through the DCAF UI Engine, so there is no need to add more code in this VI.



*Figure 1.3*

Save and Close ***TCRL User Interface.vi***.

1. Open ***TCRL Host Main.vi*** Block Diagram. Delete the bottom While Loop, as we won’t need it for this exercise. Drag and Drop ***TCRL User Interface.vi*** into the Block Diagram from the Project Window. Force ***TCRL User Interface.vi*** to execute in parallel to the DCAF engine connecting as shown in Figure 1.4.

****

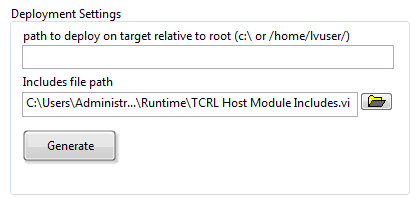
*Figure 1.4*

1. Open ***TCRL Host Main.vi*** Front Panel. In the ***configuration file path control*** browse for ***SymulatedSystem.pcfg*** located at [***\\***](file:///\\Desktop\DCAF%20Hands%20On\Temperature%20Controller\Exercise%201)***Temperature Controller***. Select this as default value for this control. Save and Close this VI.

**Part 2: Adding Required Classes**

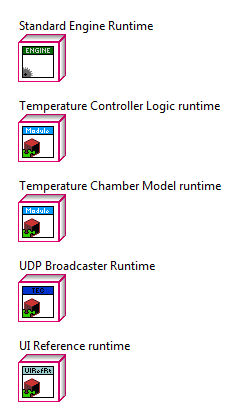
DCAF has been developed using LabVIEW Object Oriented Programming. Therefore, the code will only run if the classes used within a specific configuration are added to the project. DCAF provides a simple script that will help you with this every time you add or remove modules to a target in the Configuration Editor. This is not automatic, so you have to remember to run this scripting tool when you make these kinds of changes in the configuration.

1. Open the Standard Configuration Editor by navigating in LabVIEW to ***Tools>>DCAF>>Launch Standard Configuration Editor…***
2. Navigate within the editor to ***Tools>>Edit Plugin Search Paths***.
3. Add a search path to the plugins for this example located at ***\\Temperature Controller*** if it’s not already there.
4. In the DCAF Configuration Editor go to ***File>>Open*** and search for the ***SimulatedSystem.pcfg*** Configuration File located at[***\\***](file:///\\Desktop\DCAF%20Hands%20On\Temperature%20Controller\Exercise%201)***Temperature Controller***.
5. Take a couple of minutes to go through each component in the Simulation and UI Engines.
6. Open the Temperature Controller project located at [***\\Temperature Controller\Runtime***](file:///\\Desktop\DCAF%20Hands%20On\Temperature%20Controller\Exercise%201)if not already opened**.**
7. Open ***TCRL Host Module Includes.vi*** and verify the Block Diagram is empty. This VI will load the required classes when ***TCRL Host Main.vi*** executes. A scripting tool will add the corresponding classes to ***TCRL Host Module Includes.vi***. In the ***System Configuration*** hierarchy in the DCAF Configuration Editor select ***PC***. In the ***Includes file path*** box browse for ***TCRL Host Module Includes.vi*** located at ***\\Temperature Controller\Runtime***. Press the ***Generate*** button. Repeat this step each time you add or remove any Module from the Hierarchy Tree.



*Figure 1.5*

1. Verify that the corresponding classes have been added to ***TCRL Host Module Includes.vi*** and compare them to Figure 1.6:



*Figure 1.6*

1. Save and close ***TCRL Host Module Includes.vi***.

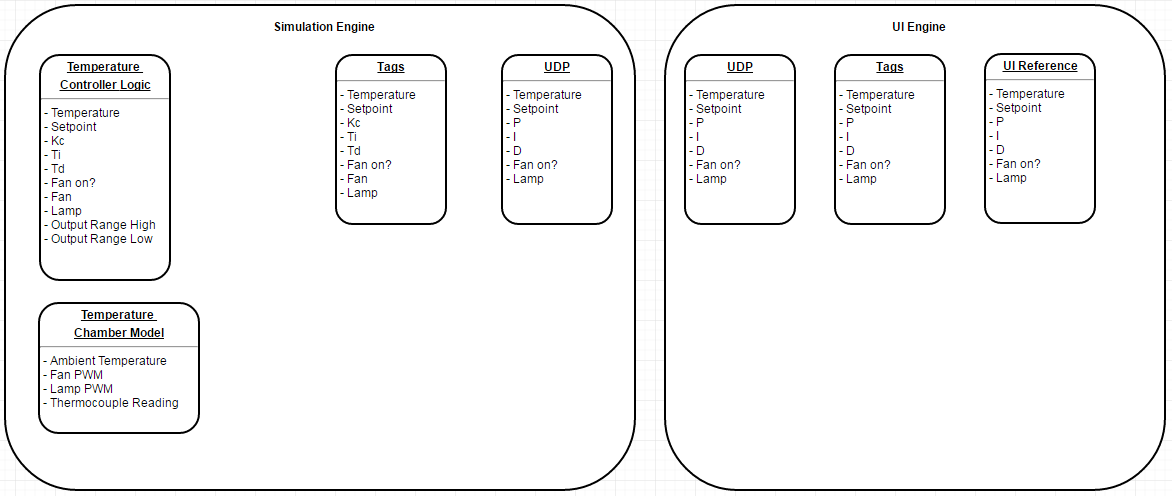
**Part 3: Mapping Tags in the Configuration Editor**

This DCAF project has 2 engines: ***The Simulation Engine*** and the ***UI Engine***. Both engines have listed Tags, Mappings, and UDP items. The rest are specific modules for each engine.

* The ***Tags*** item refers to the list of tags in the Tag Bus for each engine.
* The ***Mappings*** item allows to configure and visualize the connections between each ***Module*** parameter (Input/Output) and the Tag Bus.
* The ***UDP*** item publishes tags that can be shared with another engine that might be in the same target or in a different one.

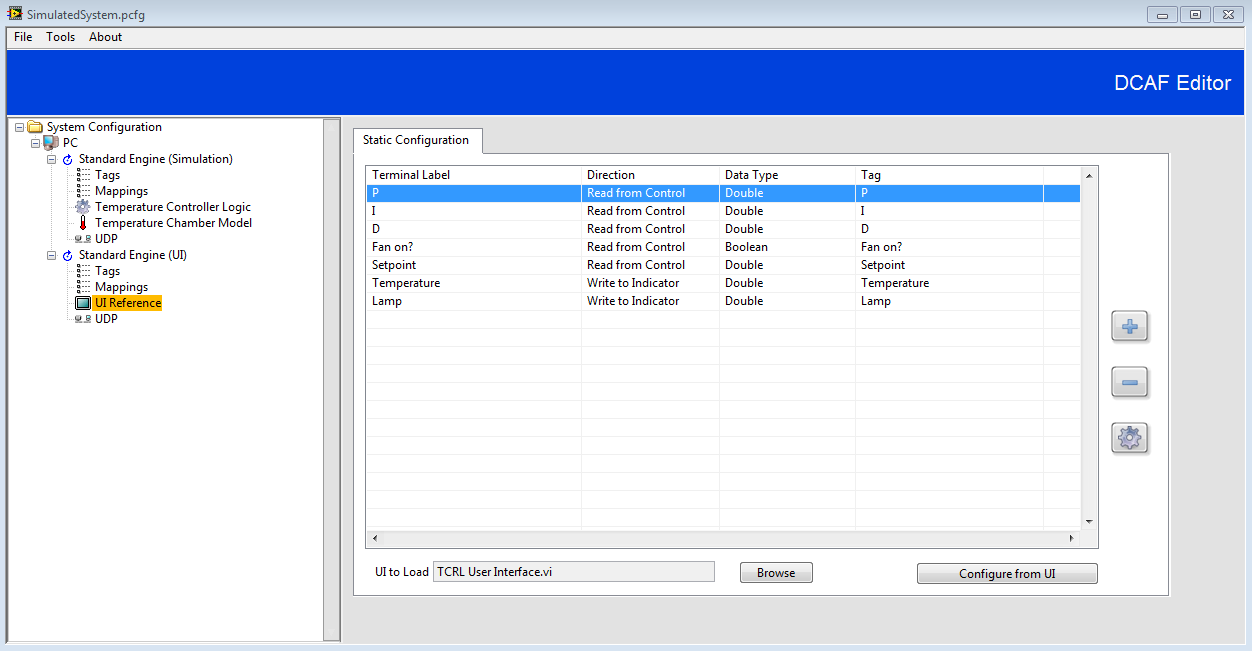
During this last part of the exercise you will learn how to map ***Tags*** between ***Modules*** through the ***Tag Bus*** in each ***Engine*** and share ***Tags*** between ***Engines*** through UDP.

Before we start the implementation, take a look to the following diagram to understand how data flows through ***Modules*** and ***Engines*** of our ***Simulated Temperature Controller***.



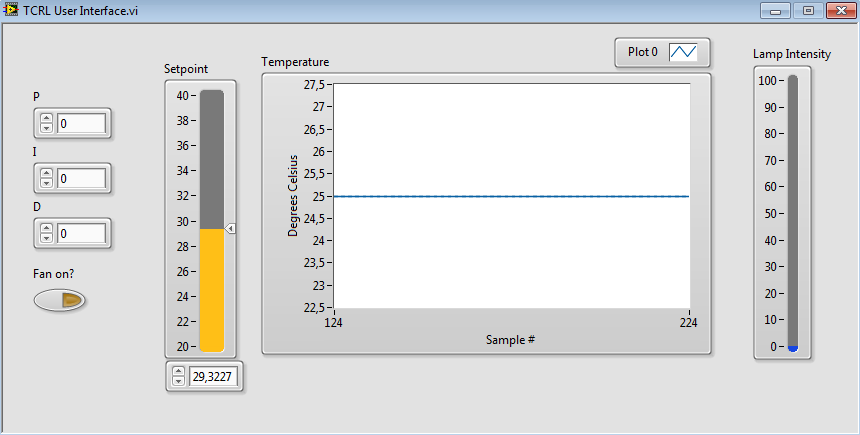
*Figure 1.7*

1. Open the Standard Configuration Editor by navigating in LabVIEW to ***Tools>>DCAF>>Launch Standard Configuration Editor…***
2. We will first map our UI to the ***UI Engine Tags***. Beneath the ***UI Standard Engine*** select ***UI Reference***. Notice the table in the ***Static Configuration*** tab is empty. Press the ***Browse*** button next to the ***UI to Load*** textbox. Browse for ***TCRL User Interface.vi*** located at [***\\Temperature Controller\***](file:///\\Desktop\DCAF%20Hands%20On\Temperature%20Controller\Exercise%201)***Runtime.***
3. Press the ***Configure from UI button***. When the pop up asking to ***Automatically map tags to channels*** appears select ***Yes***. Verify your mappings comparing them with Figure 1.8.



*Figure 1.8*

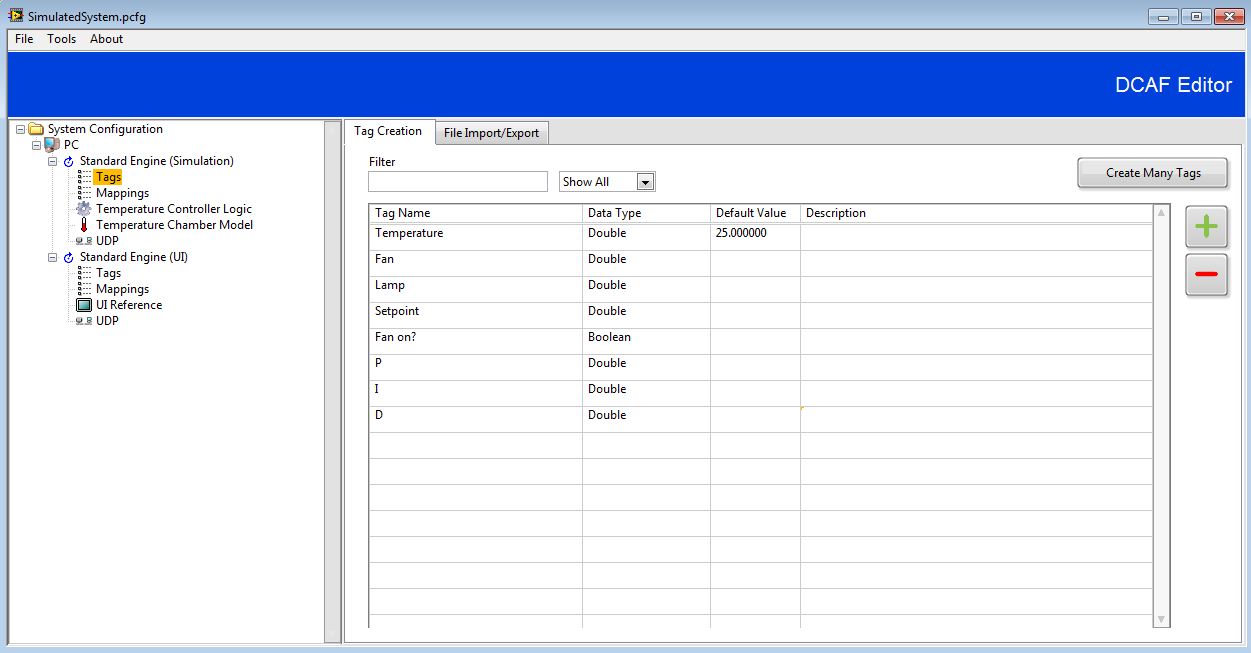
1. Save the changes in the Configuration Editor.
2. Open and run ***TCRL Host Main.vi***. Try changing the Setpoint and the other controls. Do you see any change in the temperature value displayed in the Graph?



*Figure 1.9*

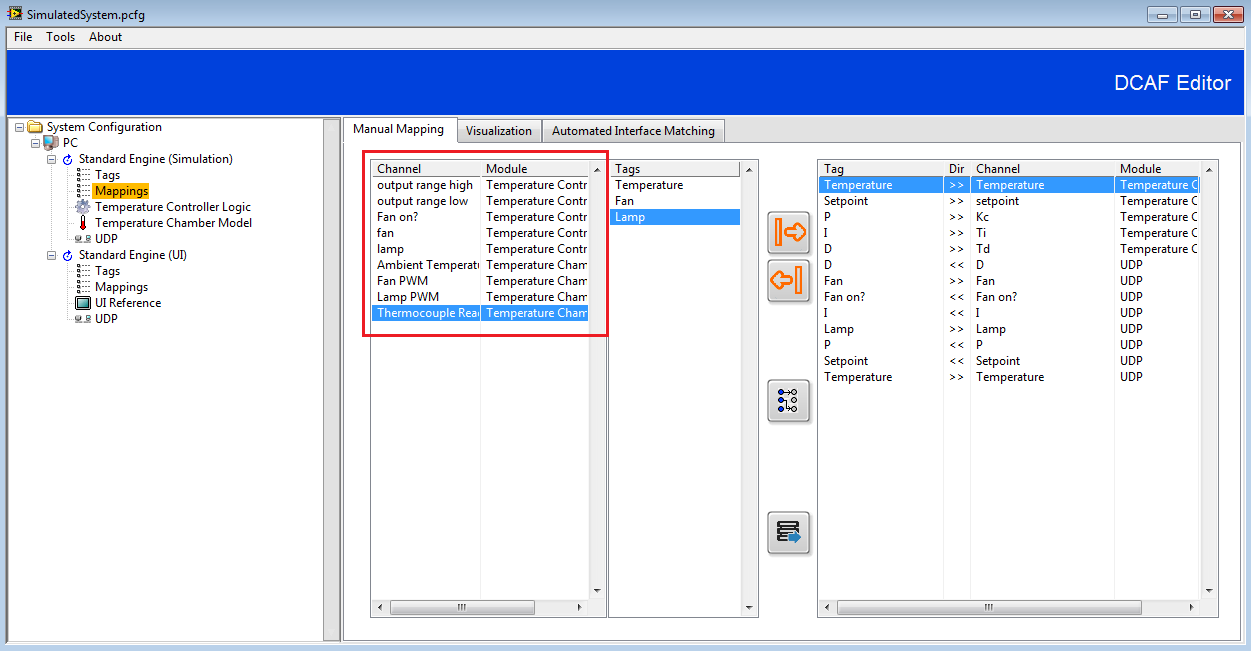
You shouldn’t see any change in the signal since we only connected the tags in the UI Engine. There are still some tags in the Simulation Engine that we need to map so we can see the PID standard behavior.

1. Stop the ***TCRL Host Main.vi*** and return to the Configuration Editor. We will review the connections in each component on both engines to understand the tag dataflow and connect the tags that are missing to make it run.
2. We will start with the Simulation Engine. First select the Tags node and take a look at the tags.



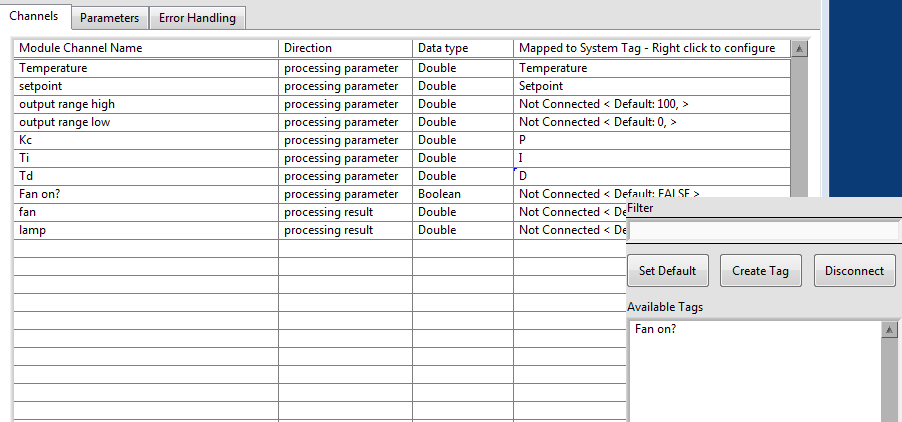
*Figure 1.10*

1. These tags are used for connections in the rest of Simulation Engine modules: Temperature Controller Logic, Temperature Chamber Model and UDP. Notice all of them are Doubles except for ***Fan on?****.*
2. Go to ***Mappings*** under the Simulation Standard Engine and select the ***Manual Mapping*** tab. This section will allow you to have a better look of the tag flow in this application. In the left pane you will see all the channels that haven’t been mapped. Just look, don’t make changes.



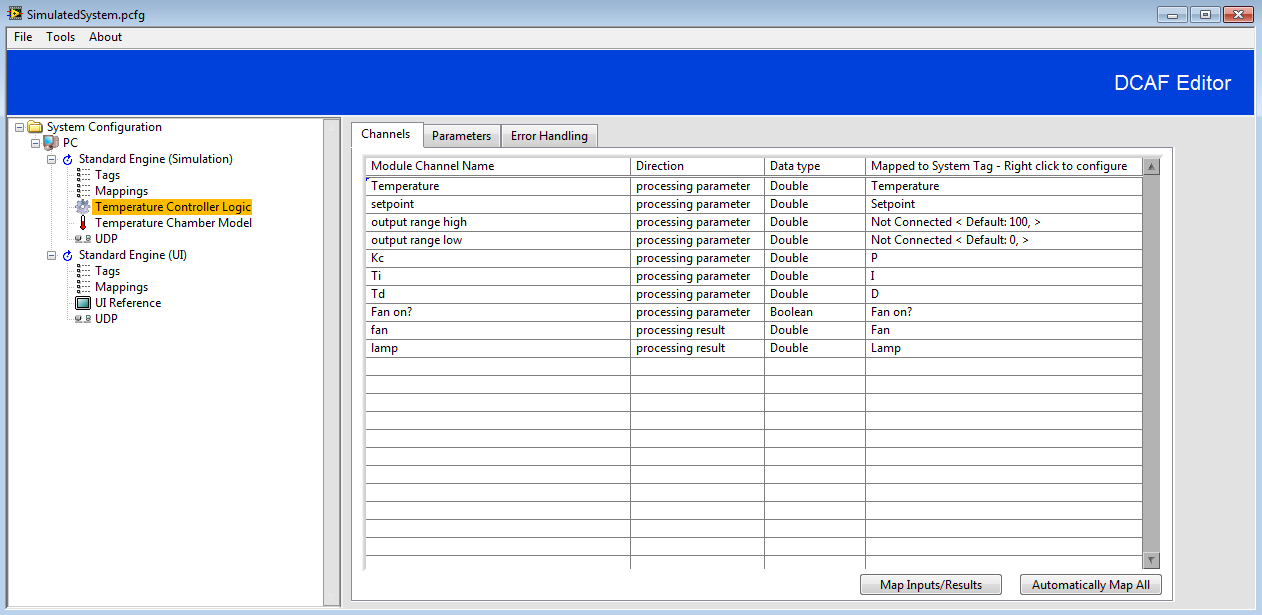
*Figure 1.11*

1. Go to the Temperature Controller Logic Module. Notice there are two variables that don’t appear in the Tag list: ***output range high*** and ***output range low***. These are internal variables with constant values defined statically. The rest should be mapped to a tag.
2. The last 3 channels should be connected to a tag (***Fan on?***, ***fan***, and ***lamp***). To connect a channel to a tag, take the cursor to the corresponding cell in the ***Mapped to System Tag*** column, left click, and select the corresponding tag from the ***Available Tags*** list.



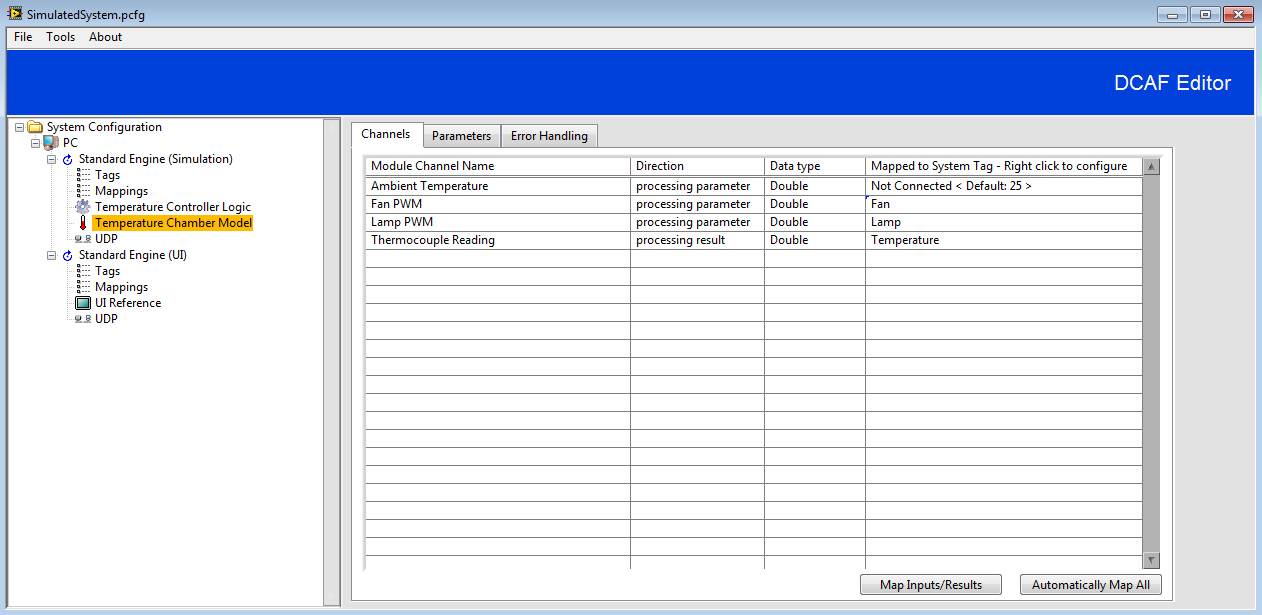
*Figure 1.12*

1. Verify your table looks like the following image:



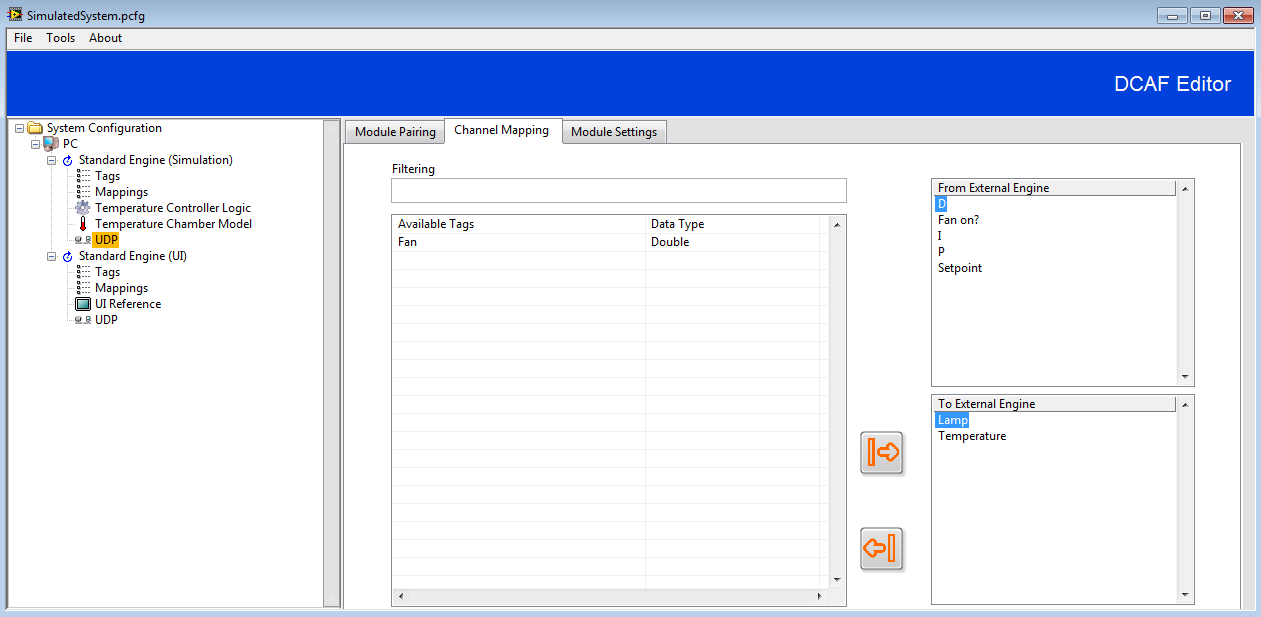
*Figure 1.13*

1. Before going to the next module notice the ***Direction*** column. ***Processing parameters*** are module inputs while ***processing results*** are module outputs. Some of the ***processing parameters*** in this module come from the ***UI Engine*** and others come from the ***Temperature Controller Logic Module***. The two ***processing results*** in this module will go through the ***Tag Bus*** as inputs in the ***Temperature Chamber Model*** module.
2. Go to the ***Temperature Chamber Model*** module. Notice all the channels are disconnected from any tag. The only disconnected channel should be ***Ambient Temperature***. Create the following connections. ***Fan PWM*** and ***Lamp PWM*** channels are ***processing parameters*** in this module that should come from *the* ***Temperature Controller Logic Module***. ***Thermocouple Reading*** is a ***processing result*** that should be used as the feedback signal in the ***Temperature Controller Logic Module*** and will also be sent to the ***UI Engine*** to be displayed in the graph. Following the same instructions as in step 11, map ***Fan PWM***, ***Lamp PWM***, and ***Thermocouple Reading*** channels to ***Fan***, ***Lamp***, and ***Thermocouple*** tags. Verify your table looks like the following image:



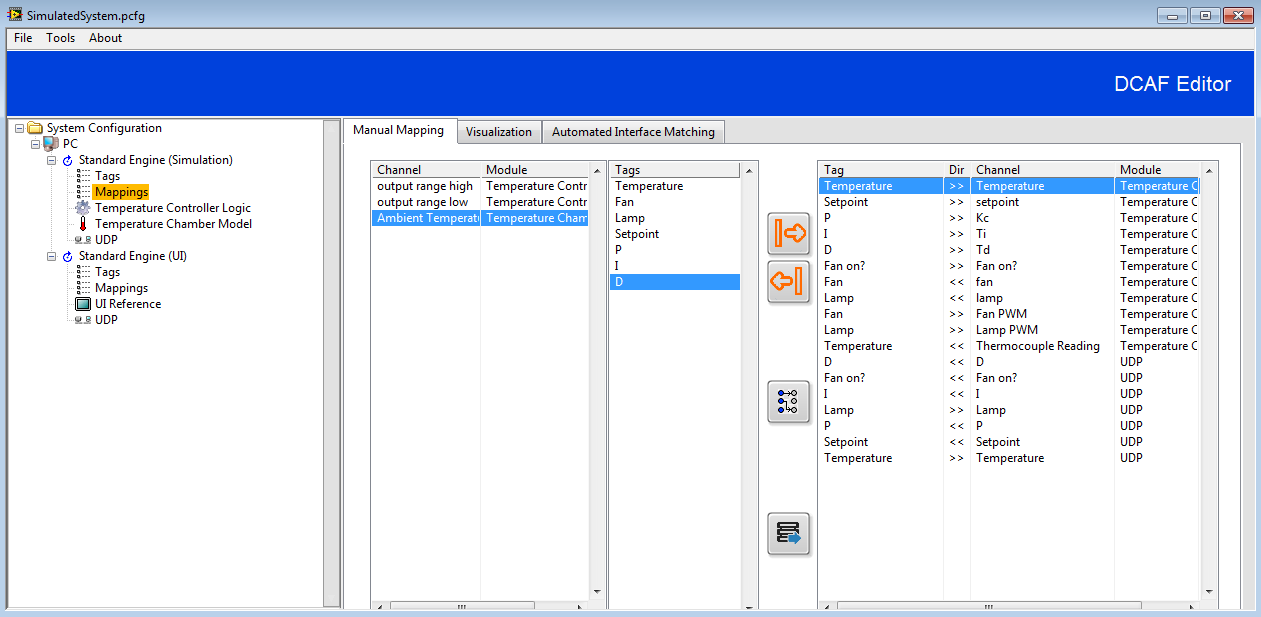
*Figure 1.14*

1. Go to the ***UDP Module*** in the ***Simulation Engine***. Go to the ***Channel Mapping Tab***. Notice the tags in the ***From External Engine*** (Inputs) and ***To External Engine*** (Outputs) boxes. Notice the ***Fan*** tag is still as an ***Available Tag***. There is no need to move it since it is not needed in the ***UI Engine,*** it is only used internally in the ***Simulation Engine.***



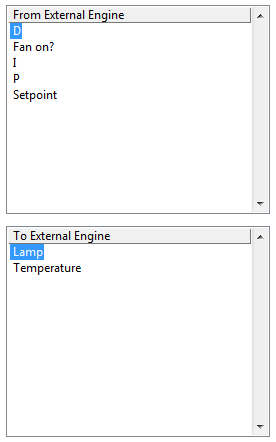
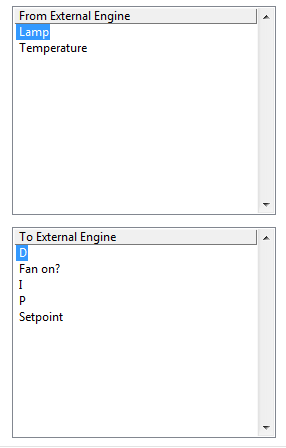
*Figure 1.15*

1. Go back to ***Mappings*** in the ***Simulation Engine***. Notice now there are only 3 channels that haven’t been mapped. There are no tags for those channels since they are configured statically in their corresponding modules or set as default. All the channels that originally were unmapped now appear mapped in the right pane. Take some time to review the mapping directions to have a better understanding of the data flow.



*Figure 1.16*

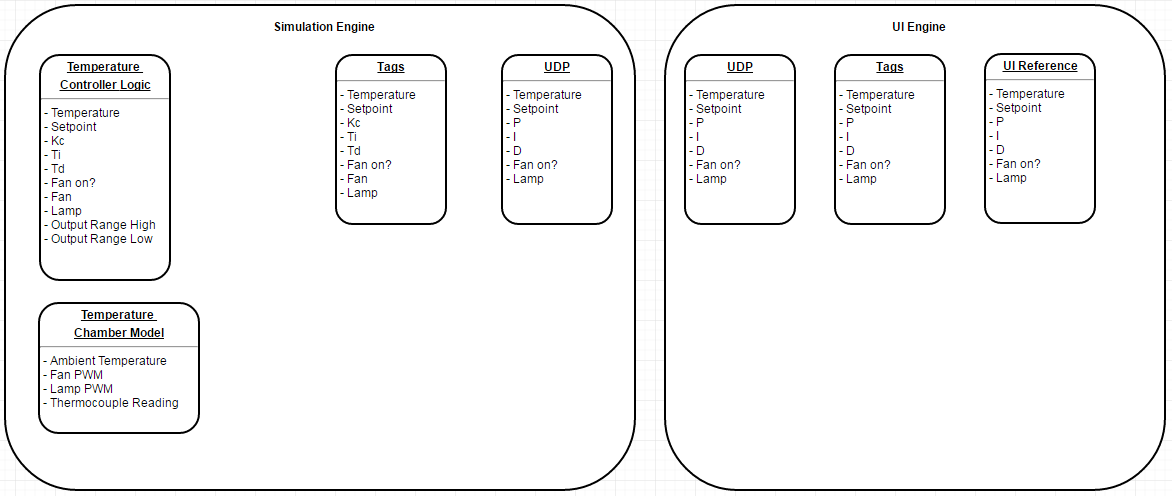
1. Now that you have seen how the ***Simulation Engine*** works, step into the different components of the ***UI Engine*** to understand how it interacts with the ***Simulation Engine***. Notice that the Inputs for the ***UI Engine UDP Module*** are the Outputs for the ***Simulation Engine UDP Module*** and vice versa.

Simulation Engine UDP UI Engine UDP

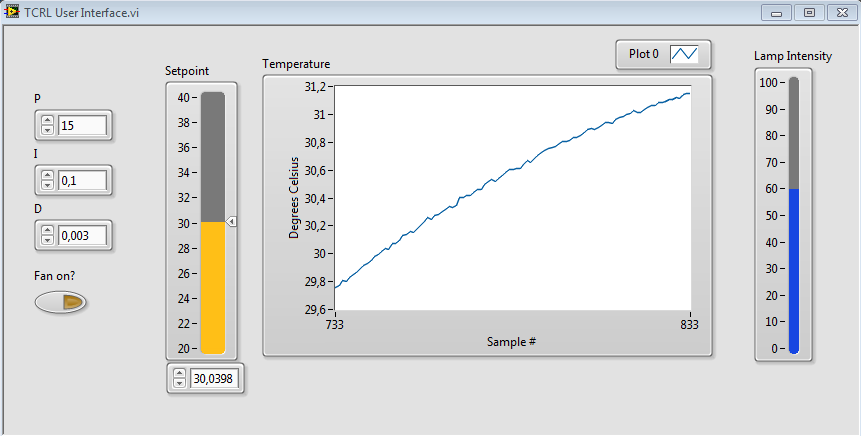
*Figure 1.17*

1. Take a look again to the dataflow diagram to review the mapping you just did.



*Figure 1.18*

1. Go to ***File >> Save*** and close the ***Configuration Editor***.
2. Open the ***Temperature Controller Example Project*** if not already open. Open and run ***Host Main.vi***.
3. Modify the ***Setpoint*** and the other controls in the ***UI***. You should now see the temperature being controlled by the ***Simulation Engine***.



*Figure 1.19*

***End of Exercise 1***

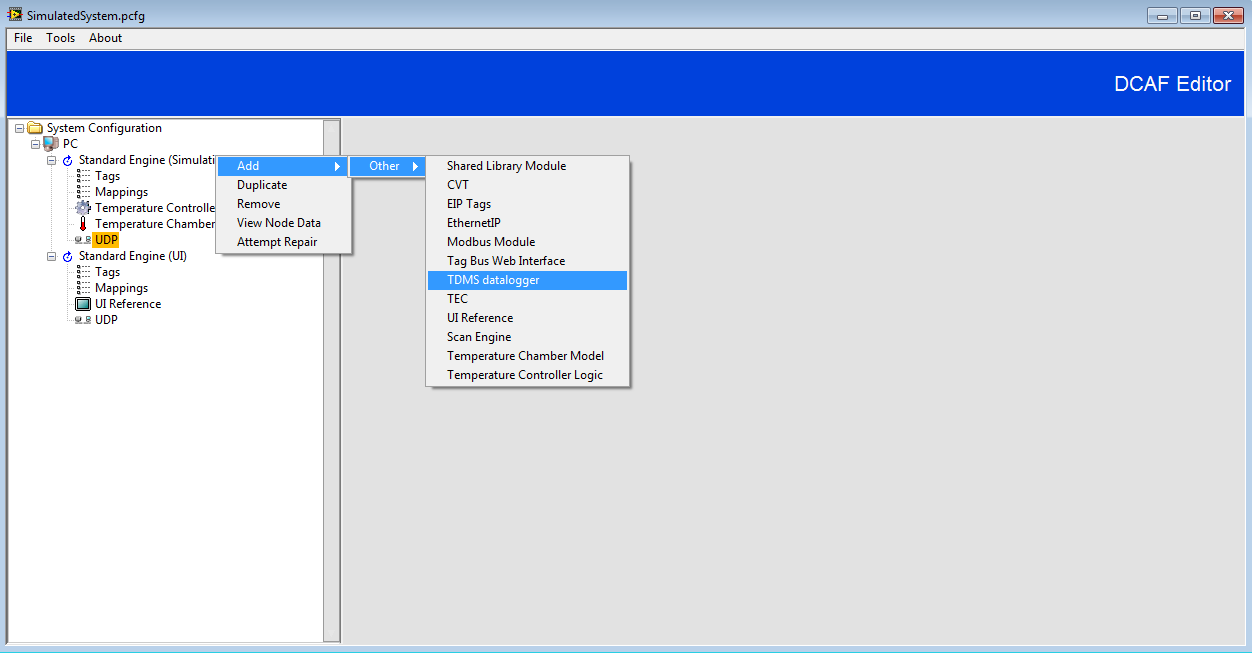
# Exercise 2: Adding Standard Modules to the Temperature Control Application (TDMS & CVT)

In Exercise 1 you developed a Simple Temperature Control Application using DCAF. Now, we will add standard features such as TDMS and CVT to learn how to add standard DCAF modules to your application.

**Part 1: Add TDMS**

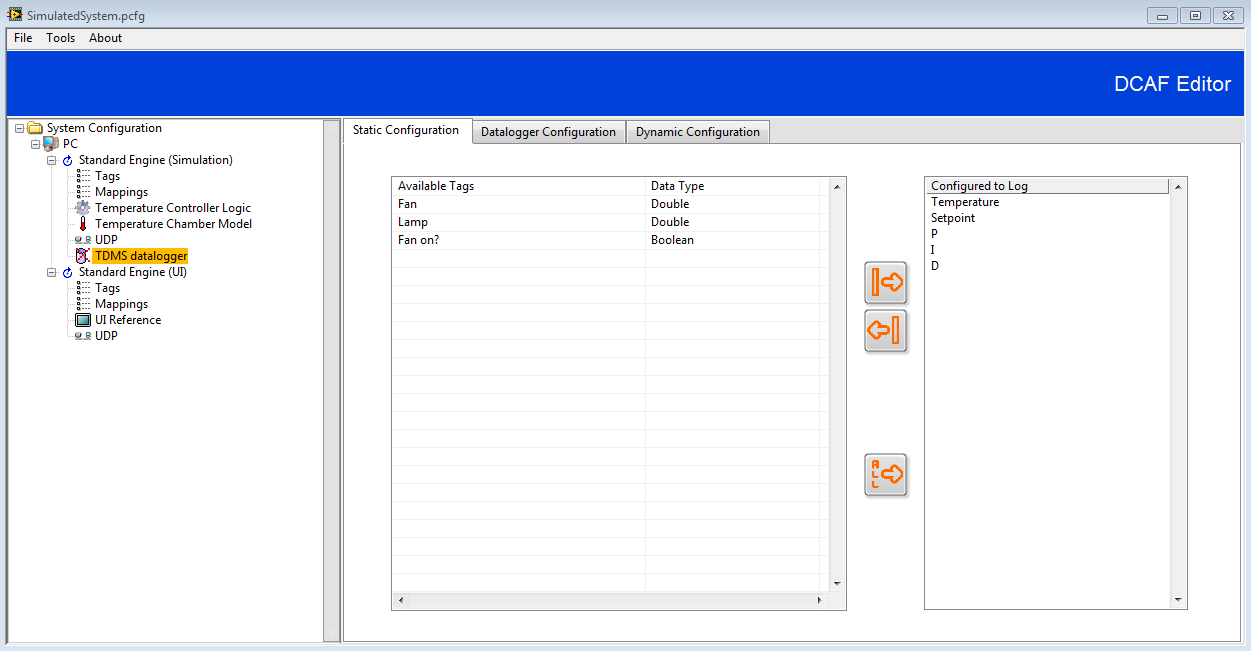
Adding TDMS is a specific module that might become really handy in a DCAF application. This part of the exercise will guide you through the process of adding TDMS logging to your DCAF application.

1. Open the ***Temperature Controller*** project you developed in Exercise 1 if not already opened.
2. Open the ***Configuration Editor*** and load ***SimulatedSystem.pcfg*** if not already opened.
3. Right click the ***Simulation Engine*** and select ***Add>>Utilities>>TDMS datalogger*** as shown in Figure 2.1.



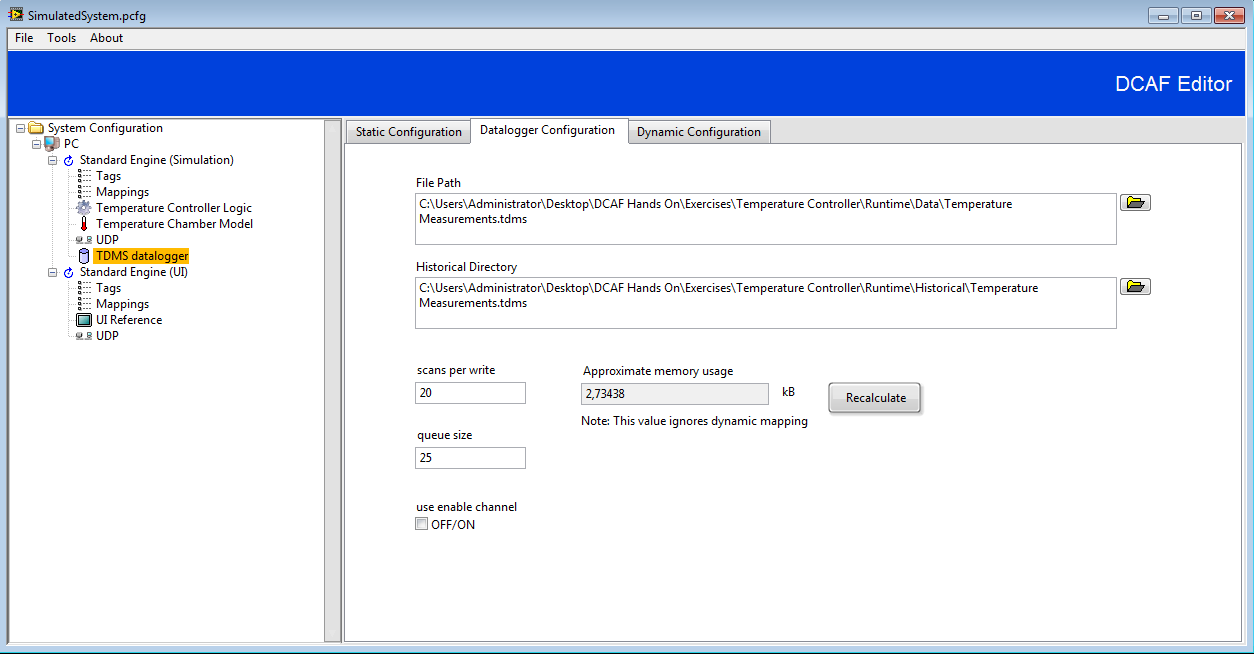
*Figure 2.1*

1. Select the ***TDMS datalogger*** item you just created. In the Static Configuration tab move Temperature, Setpoint, P, I, and D to the Configured to the ***Configured to Log*** box.



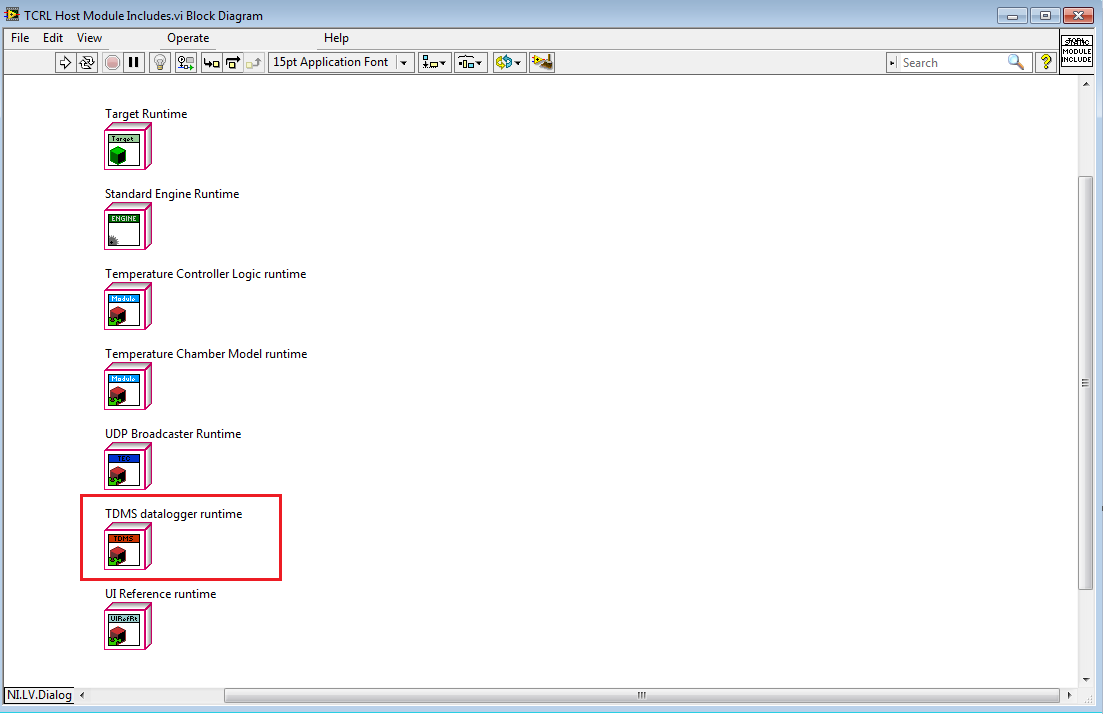
*Figure 2.2*

1. Go to the ***Datalogger Configuration*** tab. Press the first browse button to select a File path. Browse to **\\Temperature Controller\Runtime** and create a folder named ***Data***. Type ***Temperature Measurements*** as the ***File name***.
2. Press the second browse button to select a Historical Directory. Browse to **\\Temperature Controller\Runtime** and create a folder named ***Historical***. Type ***Temperature Measurements*** as the ***File name***. Verify your file paths with Figure 2.3.



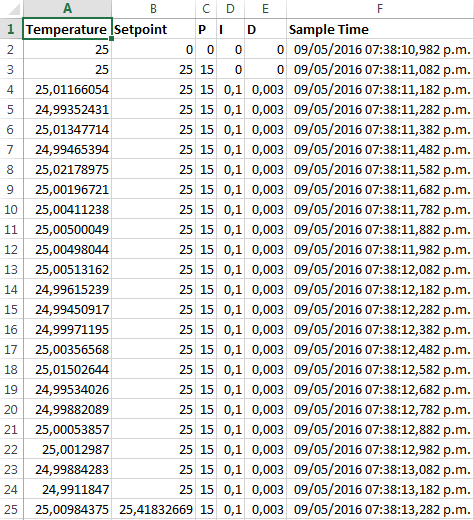
*Figure 2.3*

1. Since we added a new module, loaded classes should be updated. Use the scripting tool explained in ***Exercise 1>>Part 2>>Step 7*** to update the classes in **TCRL Host Module Includes.vi**. Verify the class has been successfully added to **TCRL Host Module Includes.vi.**



*Figure 2.4*

1. Run TCRL Host Main.vi. Do some changes to the setpoint and verify it still working and stop the VI.
2. Go to the Historical folder you created located at **\\Temperature Controller\Runtime** and open the TDMS file just created. Verify the tags you added in the TDMS datalogger modules appear in the file and generated data.



*Figure 2.5*

**Part 2: Add CVT**

Sometimes you will need to share tags with code that might run asynchronously in parallel with the DCAF engine. Current Value Table (CVT) is a component that provides a simple interface between DCAF and other LabVIEW code.

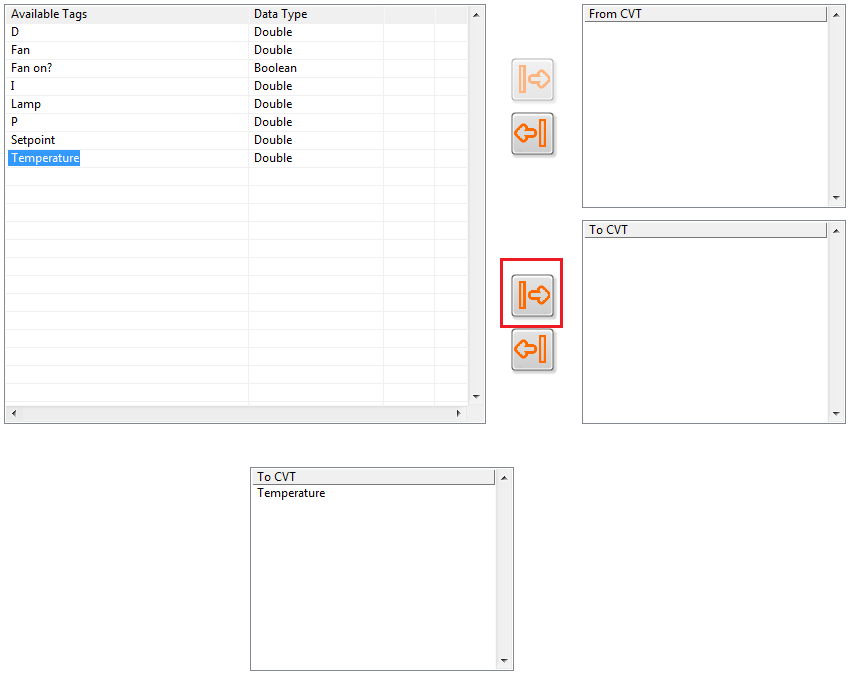
During this part of the exercise we will publish the Temperature tag and visualize it in the Front Panel of your ***TCRL Host Main.vi***.

1. Open the ***Temperature Controller*** project if not already opened.
2. Open the ***Configuration Editor*** and load ***SimulatedSystem.pcfg*** if not already opened.
3. Add a CVT module to the Simulation Engine in the same way you added the TDMS datalogger.
4. Select the ***CVT*** module. Select ***To CVT*** direction.



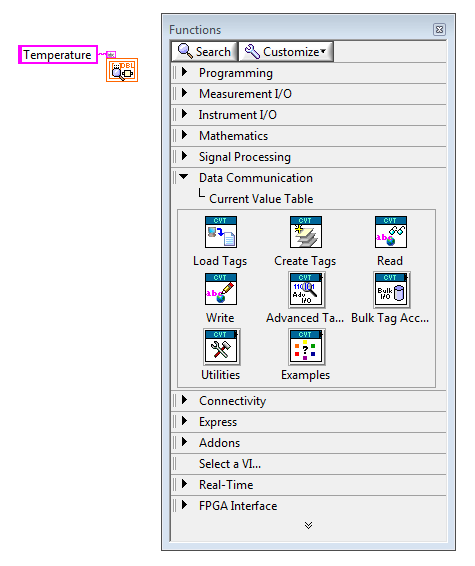
*Figure 2.6*

1. Move Temperature to the ***To CVT box***.



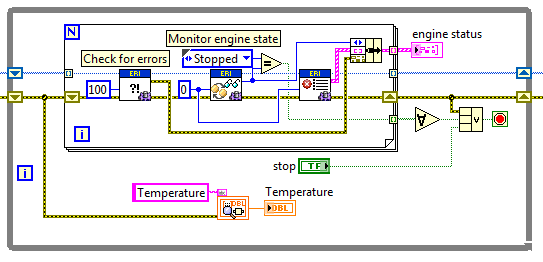
*Figure 2.7*

1. Save your configuration.
2. Open ***TCRL Host Main.vi*** Block Diagram. Add a ***Read*** VI from the ***Current Value Table Function Palette***. By default it is a double. Connect a string constant to the ***Tag Name Terminal*** and type ***Temperature***.



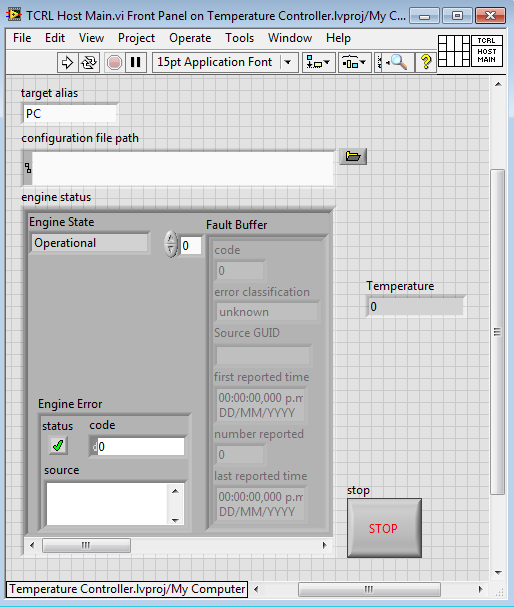
*Figure 2.8*

1. Finish the code as shown in Figure 2.9



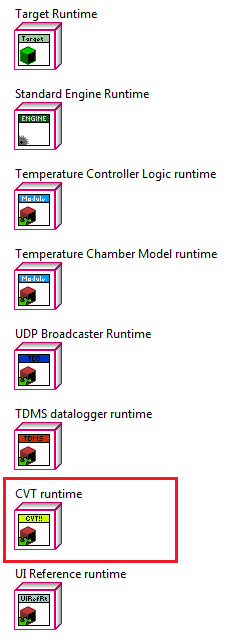
*Figure 2.9*

1. Rearrange the front panel Indicators such that the new Temperature indicator is visible as shown in Figure 2.8.



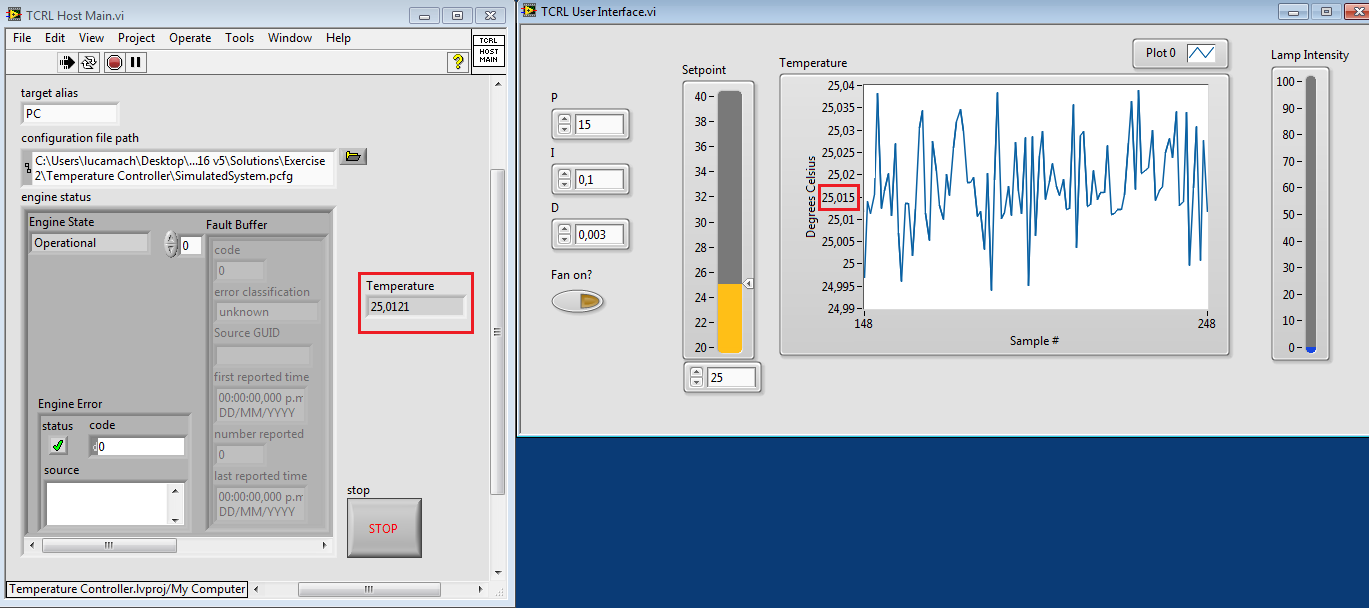
*Figure 2.10*

1. Save the changes in TCRL Host Main.vi. Go back to the Configuration Editor and update the classes for TCRL Host Module Includes.vi as you did for the TDMS Datalogger Module. Verify the CVT class is added to TCRL Host Module Includes.vi.



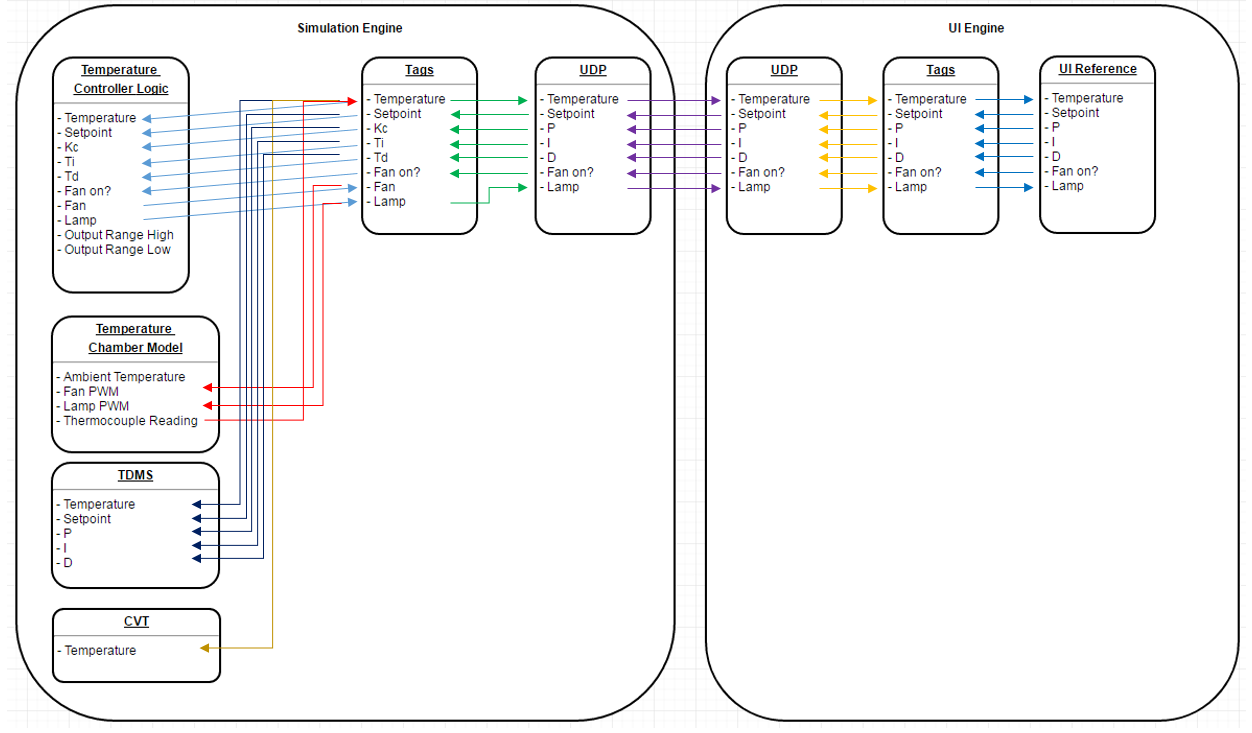
*Figure 2.11*

1. Run TCRL Host Main.vi. The UI should still be working. Verify the value displayed in the new Temperature indicator in TCRL Host Main.vi corresponds to the value displayed in the Temperature chart in TCRL User Interface.vi



*Figure 2.12*

1. Stop and close TCRL Host Main.vi.
2. Take a look to the following diagram to verify the updated mappings.



*Figure 2.13*

***End of Exercise 2***

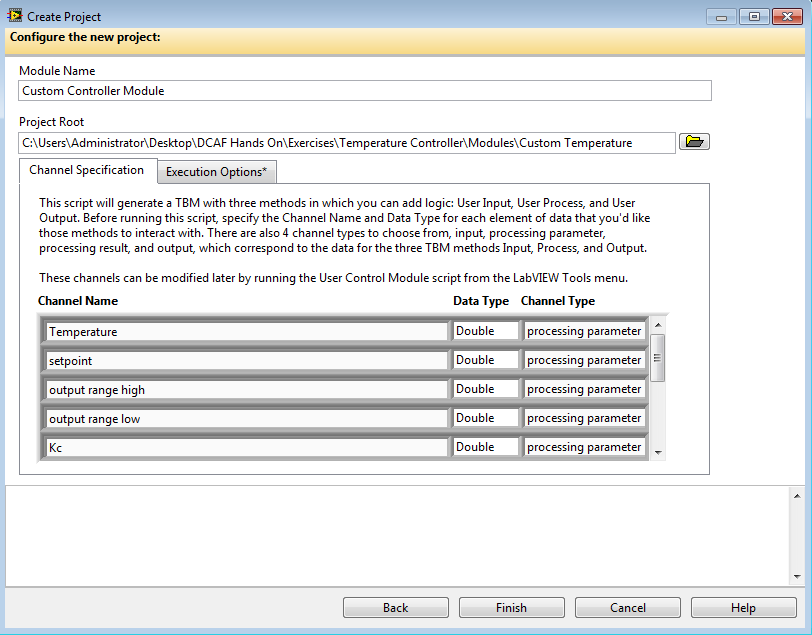
# Exercise 3: Creating a Custom Module

In this exercise we will implement a custom control algorithm for our temperature controller. This can be actually custom, or a simple PID controller as is used by the original example.

1. Navigate to ***Project >> Create Project…***
2. In the tree on the left, select ***DCAF >> Modules*** and select ***DCAF Static Channel Module***.
3. Type ***Custom Temperature******Controller*** as the ***Module Name***.
4. Look for the Project Root ***\\Temperature Controller\Module*** and create a folder called ***Custom Temperature Controller***. Get into that folder and press ***Current Folder***.
5. Enter a new name and project path for your module.
6. Add the following parameters to your new module as Channels in the ***Channel Specifications*** tab. Channels represent data passed to or from your module during different execution stages, and channel names are case sensitive. The direction specifies whether the data is to the module or from the module, and is divided into inputs (data provided by input.vi), outputs (data provided to output.vi), and processing parameters and results (data passed to and from process.vi). For this hands on we will implement a processing step.

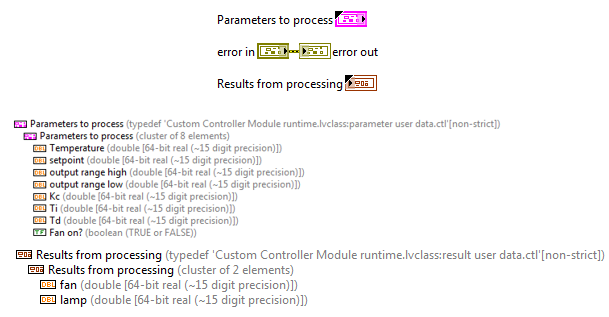
|  |  |  |
| --- | --- | --- |
| Name | Type | Direction |
| Temperature | Double | Processing parameter |
| setpoint | Double | Processing parameter |
| output range high | Double | Processing parameter |
| output range low | Double | Processing parameter |
| Kc | Double | Processing parameter |
| Ti | Double | Processing parameter |
| Td | Double | Processing parameter |
| Fan on? | Boolean | Processing parameter |
| fan | Double | Processing result |
| lamp | Double | Processing result |

Your configuration should look like the one shown in Figure 3.1.



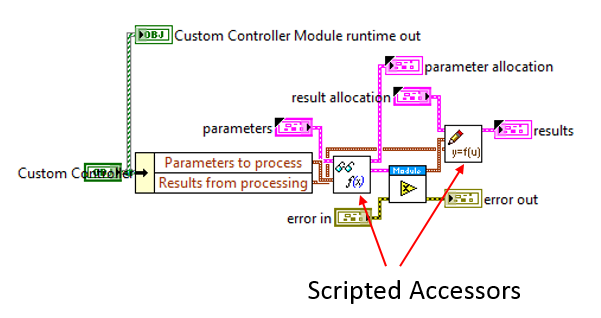
*Figure 3.1*

1. Press **Finish**.
2. Your new project will appear. Navigate to **Custom Controller Module runtime.lvclass** and open ***user process.vi***. This method should have two clusters, one input and one output, which match the list of tags above. This clusters were created based on the table you just filled in.



*Figure 3.2*

1. Open ***process.vi*** in the overrides folder.

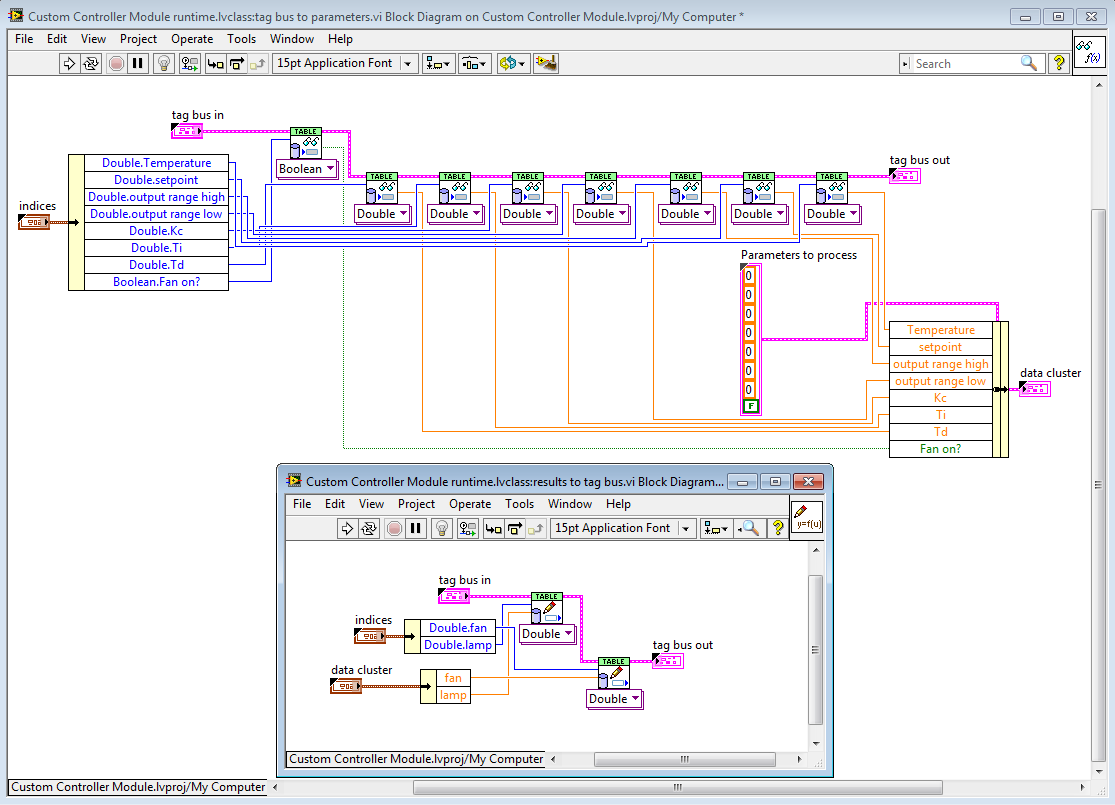


*Figure 3.3*

On either side, ***Scripted Accessors*** convert tag bus data into your user-defined cluster. These methods are *not* automatically generated by the project scripting tool and must be generated when the project is first scripted or after any change to the interface. If you need to make changes to the interface, go to ***Appendix A***.  
**Note:** During this process, the lvclass files are modified and so LabVIEW requires that your classes are not loaded in multiple contexts. That is, you must close any other projects which currently have the class loaded. If you see a “lock” icon over the class, the script will not work. To script these methods:

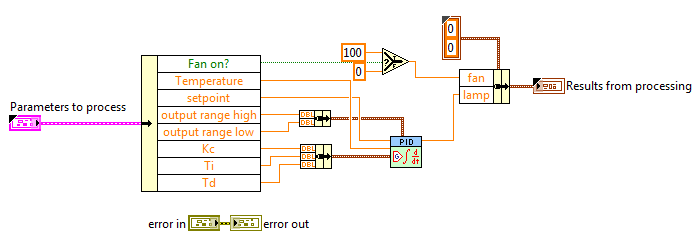
* 1. Open ***Tools >> DCAF >> Launch Control Module Scripting Utility…***
  2. Drag the runtime class from the project over the runtime class path control or browse for it manually, then repeat for the configuration class (*YourModuleName* Configuration.lvclass).
  3. Because you used the script, you can leave most options as the default and press **Run**.

1. The VIs that were scripted should look like the ones in Figure 3.4.



*Figure 3.4*

1. Return to ***process.vi***. The function being called in the middle is ***User process.vi***, and is the function you will implement. You will implement the same PID control that the ***Temperature Controller Module*** used in previous exercises.

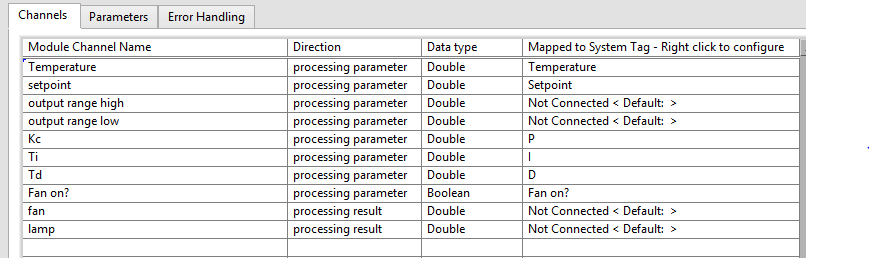


*Figure 3.5*

Follow the next steps to implement the code shown in Figure 3.5:

* 1. Drop down an instance of ***PID.vi*** from ***Control and Simulation >> PID >> PID.vi***.
  2. Drop down the following functions: 1 ***Unbundle by Name***, 2 ***Bundle*** functions, 1 ***Bundle by Name*** function, and a ***Select*** function.
  3. Wire the ***Unbundle by Name*** function to the ***Parameters to process*** cluster. Expand all the terminals.
  4. Wire ***Parameters to process.Temperature***to ***PID.vi*** ***process variable*** input.
  5. Wire ***Parameters to process.setpoint***to ***PID.vi*** ***setpoint*** input.
  6. Bundle ***Parameters to******process.output range high*** *and* ***Parameters to******process.output range low*** and wire the cluster to ***PID.vi output range*** input
  7. Bundle ***Parameters to******process.Kc, Parameters to******process.Ti,*** *and* ***Parameters to******process.Td*** and wire the cluster to ***PID.vi PID gains*** input
  8. Create a constant from ***Results from processing*** indicator and connect it to the ***Bundle by Name*** ***input cluster*** terminal. Expand the ***Unbundle by Name*** Function to have 2 terminals
  9. Wire ***Parameters to process.Fan on?*** to the ***Select*** function and set the two values as ***t=100*** and ***f=0***. Wire the output of this function to the ***Results from processing.fan value***
  10. Wire ***PID.vi output*** to ***Results from processing.lamp***.
  11. The result should look something like Figure 3.5.

1. Save the new project and close it.
2. Reopen ***Temperature Controller Example.lvproj*** that you made in the earlier exercises (if you closed it), and reload the Standard Configuration Editor (Open ***Tools >> DCAF >> Launch Standard Configuration Editor…***).
3. Navigate to ***Tools >> Edit Plugin Search Paths***.
4. Press **Add** and navigate to the location of your new control module (***\\Temperature Controller\Module\Custom Temperature Controller***).
5. Reopen ***<LabVIEW>\examples\TBD Examples\RT Temperature Controller\SimulatedSystem.pcfg*** and then to go ***File >> Save As…*** to make a copy of the configuration. For simplicity, save it in the same location but call it ***SimulatedSystemCustomControl.pcfg***.
6. Now, **right click** on ***Standard Engine (Simulation)***and select **Add >> Other>>*Custom Controller Module***. Then, select this new module from the tree.
7. For each ***processing parameter*** tag, right click on the column ***Mapped to System Tag*** and configure the channel to be mapped to the appropriate system tag. You can look at “Temperature Controller Logic” to identify the correct mapping, which looks like Figure 3.6:



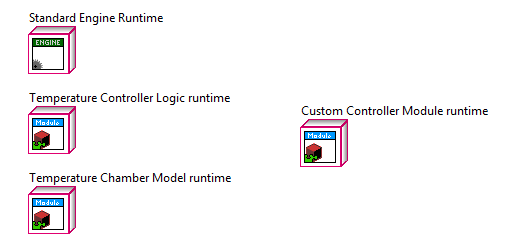
*Figure 3.6*

1. Now, click on the output ranges and press **Set** **Default** for each. Set output range high to 100 and output range low to 0 (this is not required, as the default if not set is 0). These are unmapped parameters, meaning they can be modified in the editor but they are constants at runtime – output range high will always be 100.



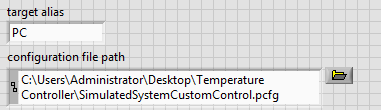
*Figure 3.7*

1. Finally, we need to map our results to system tags. However, the editor prohibits mapping multiple writer channels (***processing result*** or ***output***) to a single system tag – mapping two writers to a single tag would introduce a race condition. So the next step is to either remove the mappings from the ***Temperature Controller Logic*** ***Module***, or to delete the module entirely. In either case the write reservations will be released. Reader channels may be ignored, as any tag can be read from multiple channels.
   1. Select ***Temperature Controller Logic***, right click on rightmost column for the ***fan*** and ***lamp*** channels, and select **disconnect**.
   2. **And/Or:** Right click on ***Temperature Controller Logic*** in the tree and select ***Remove*.**
2. Return to the ***Custom Controller Module*** configuration screen and map channel ***fan*** to tag ***Fan*** and channel ***lamp*** to tag ***Lamp***.
3. Select to ***File >> Save*.**
4. From the ***Temperature Controller*** Project Window, open ***Host Module Includes.vi***. This function ensures that all appropriate modules are loaded into memory. You can also load precompiled modules (llb or lvlibp files) from disk, but for our purposes we will simply hardcode the appropriate modules.
5. Open the Custom Temperature Controller Project located in ***\\Temperature Controller\Module\Custom Temperature Controller***, if not already opened. Drag an instance of ***Custom Controller Module runtime.lvclass*** from the project onto the diagram of Host Module Includes.vi. This ensures that your new module is always loaded into memory. You could also have achieved the same result by clicking ***Generate*** in the Standard Configuration Editor.



*Figure 3.7*

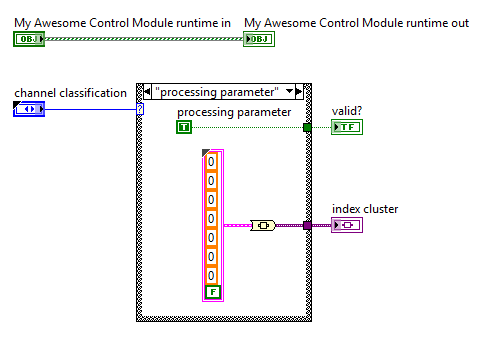
1. Now save and close this VI and return to ***Host Main.vi***. On the front panel, browse for your configuration file (we suggested ***SimulatedSystemCustomControl.pcfg***) in the configuration file path control.



1. Run ***Host Main.vi***. The behavior should match that of the original controller used in previous exercises.

***End of Exercise 3***

# Appendix A: Modifying control module data

1. If you made a mistake on your module or would simply like to add or remove different methods (for example, you want to add input functionality to your module), you will need to navigate to *get data definition.vi* in your runtime class. The “valid?” Boolean for each case of the case structure indicates whether or not the data interface is valid and if that method should be run. Setting a given Boolean to true tells the framework that you’d like to configure and run that method. If you followed the steps above, the Boolean should be true for *processing parameters* and *processing results*. The script will use these Booleans to determine if a given cluster is valid (the “placeholder” fields are just there to ensure the clusters are not broken).  
   
2. To add additional input or output channels, simply drop the appropriate controls or indicators into one of the four clusters in the *user data* virtual folder.  
   