Temperature Chamber Detailed Documentation

Introduction

This example demonstrates the implementation of a simple temperature chamber controller application using the Distributed Control and Automation Framework (DCAF). This example 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. The goal of this example is to provide a baseline understanding of how the framework works, of the benefits the framework can provide, and of the workflow required to build your own applications. This document will walk you through the steps required for getting the example up and running while pointing out framework features and terminology along the way.

The example includes two implementations. The first makes use of a simulated system with a simple model of a temperature controller that can be run without hardware. 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

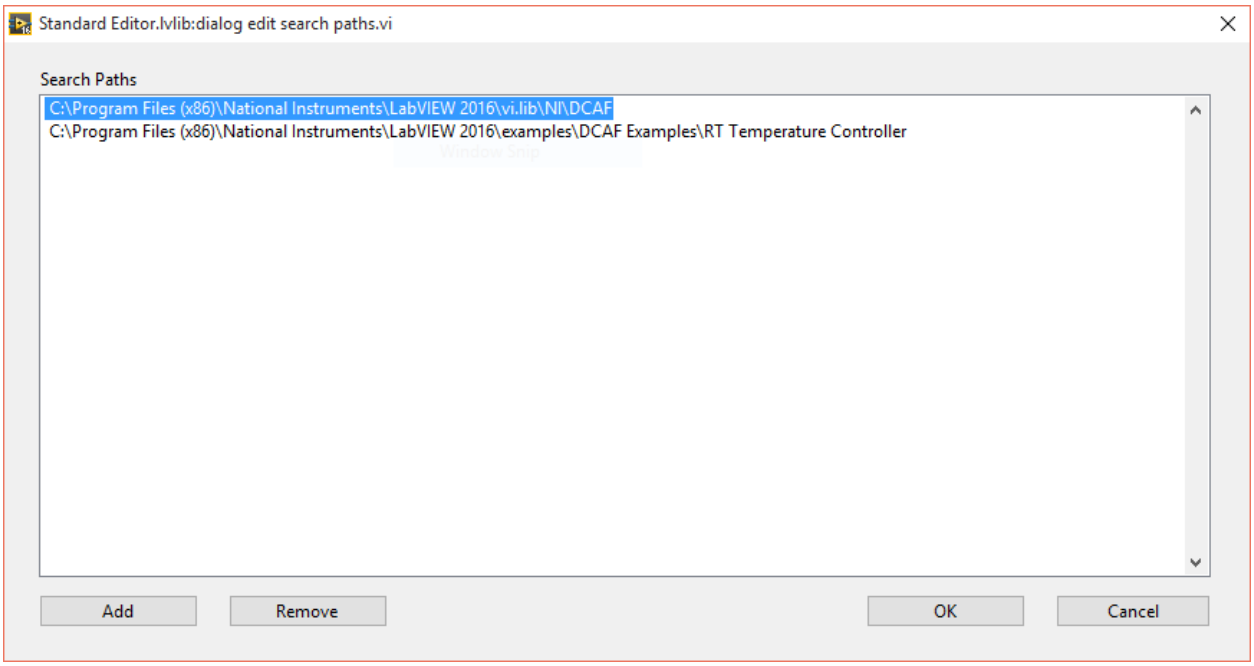
Configuration Editor

DCAF 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 DCAF implementation.

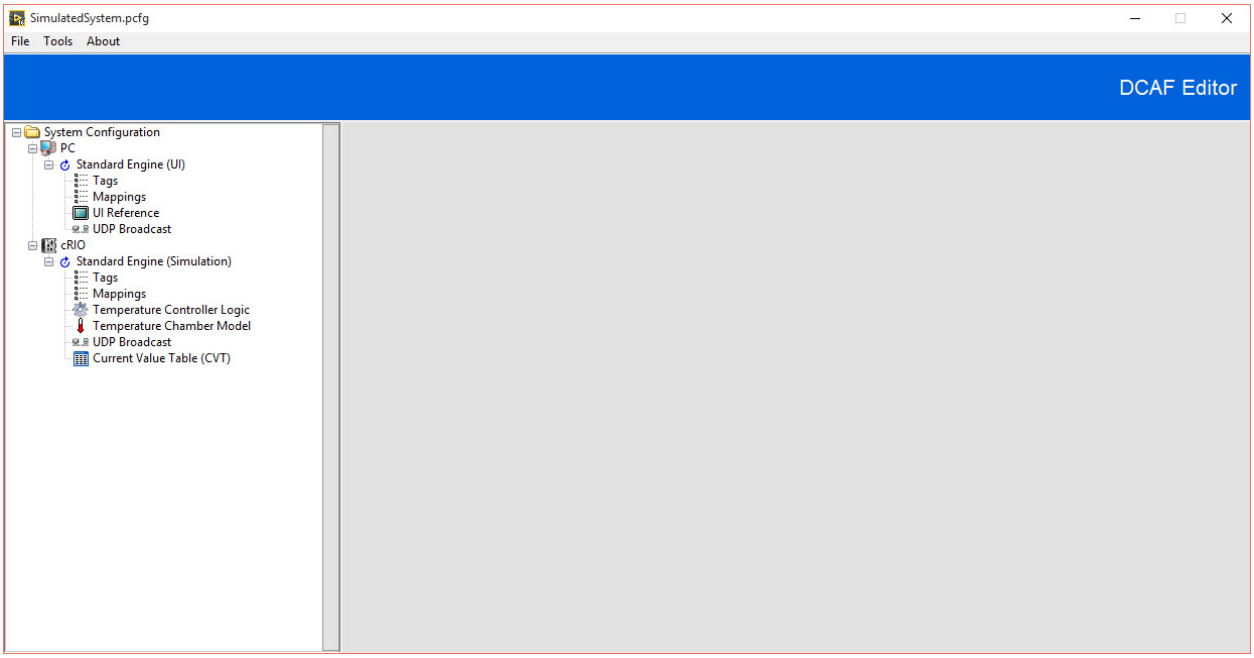
1. Open up the Standard Configuration Editor for DCAF by navigating in LabVIEW to **Tools>>DCAF>>Launch Standard Configuration Editor…**

This is the default editor for the framework and is used to view and modify system configurations. It may take a few seconds to load as it searches for and loads into memory the various framework plug-ins. Before loading up this example's configuration file, first ensure that the editor can find all of the DCAF plug-ins that are a dependency of this configuration. Any plug-ins installed to vi.lib are found by default.

2. Navigate within the editor to **Tools>>Edit Plugin Search Paths**.
3. Add a search path to the DCAF plugins for this example located at **<LabVIEW examples>\DCAF Examples\RT Temperature Controller** if it's not already there. Also confirm that the standard vi.lib file paths are specified as shown below for the version of LabVIEW that you are using.



4. Click **OK** to confirm the new search path. The configuration editor will now scan these directories for any DCAF plugins and load them into memory.
5. Once the busy cursor disappears, navigate to **File>>Open** and open up the configuration file for this example at **<LabVIEW Examples>\DCAF Examples\RT Temperature Controller\SimulatedSystem.pcfg**.



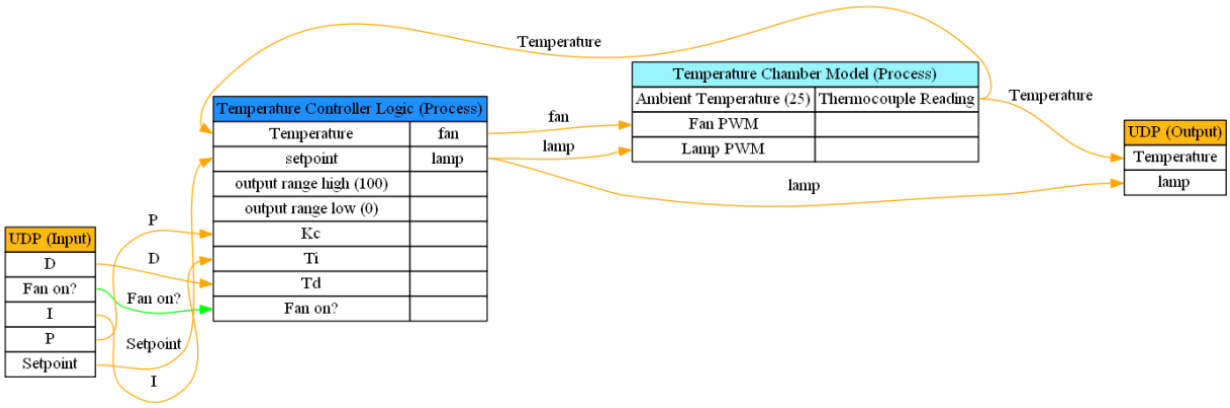
You should now see a tree control populated on the left side of the editor with a hierarchal set of items. Clicking on an item in the tree control populates the view on the right which can be used to edit that item's configuration. Feel free to explore the editor by clicking various nodes within the 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.

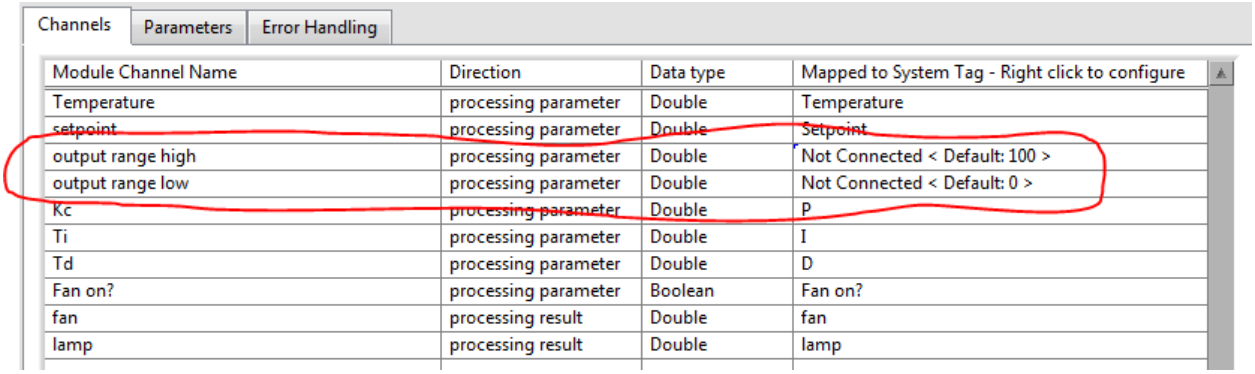
An Engine is essentially a background process with a timing source and a collection of named locally-scoped current value data (Tags). Each Engine can be configured to execute one or DCAF Modules. Mappings are used to specify the exchange of data within a DCAF module (called a Channel) with the Tag data in the engine. Two DCAF modules 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 logic). 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 DCAF modules, Channels, and Tags. In the diagram, each colored row represents a DCAF module method. The entries under the DCAF module method represent the Channels of that DCAF module. 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.



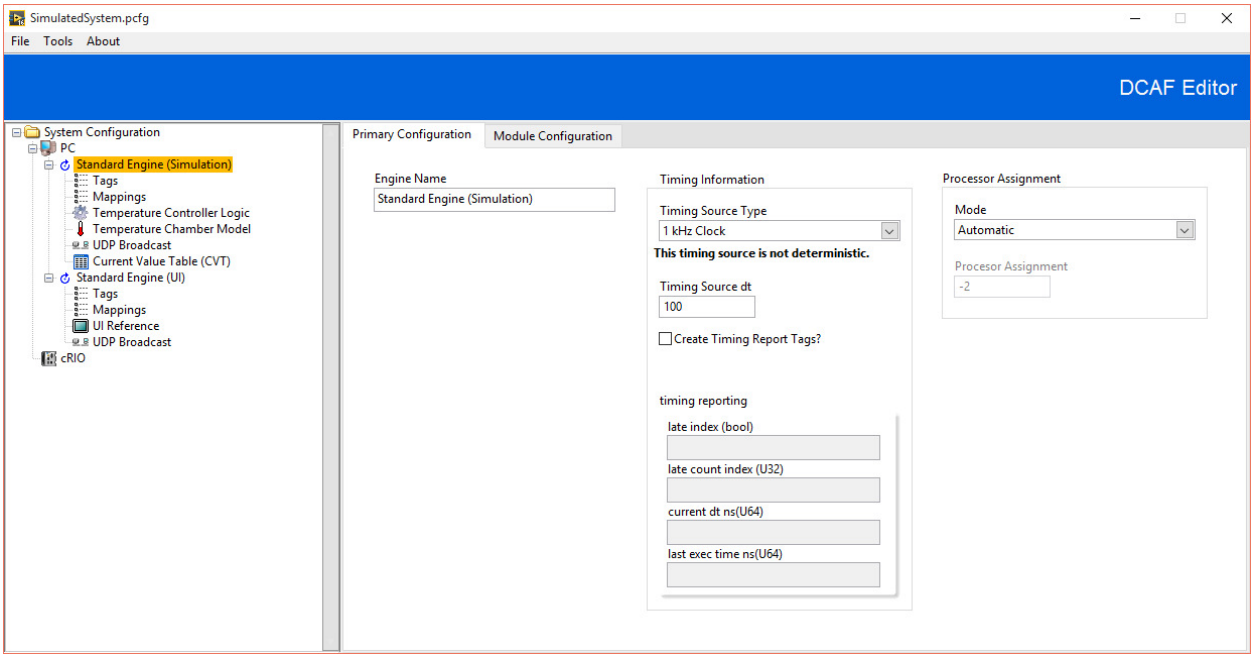
In addition to calling DCAF modules and exchanging data between them, the engine is also responsible for determining the execution rate of the system as well as the error handling configuration for each module. Click on the 'Standard Engine (Simulation)' node in the tree to see how these are configured 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. It also allows an operator to manipulate the PID gains. Status information regarding the operation of the controller is then returned and presented on this UI.

Data is transferred between the Simulation and UI engines using two instances of a UDP DCAF module. Each engine has its own instance and the two instances are paired together. All Tag data within an engine can only be accessed by DCAF modules. However a DCAF module may then expose that Tag data to another engine or to other threads running on that target. In this case the UDP DCAF module is used to reflect tag data between the two engines.

This configuration must be modified to properly reflect the IP addresses of your execution targets.

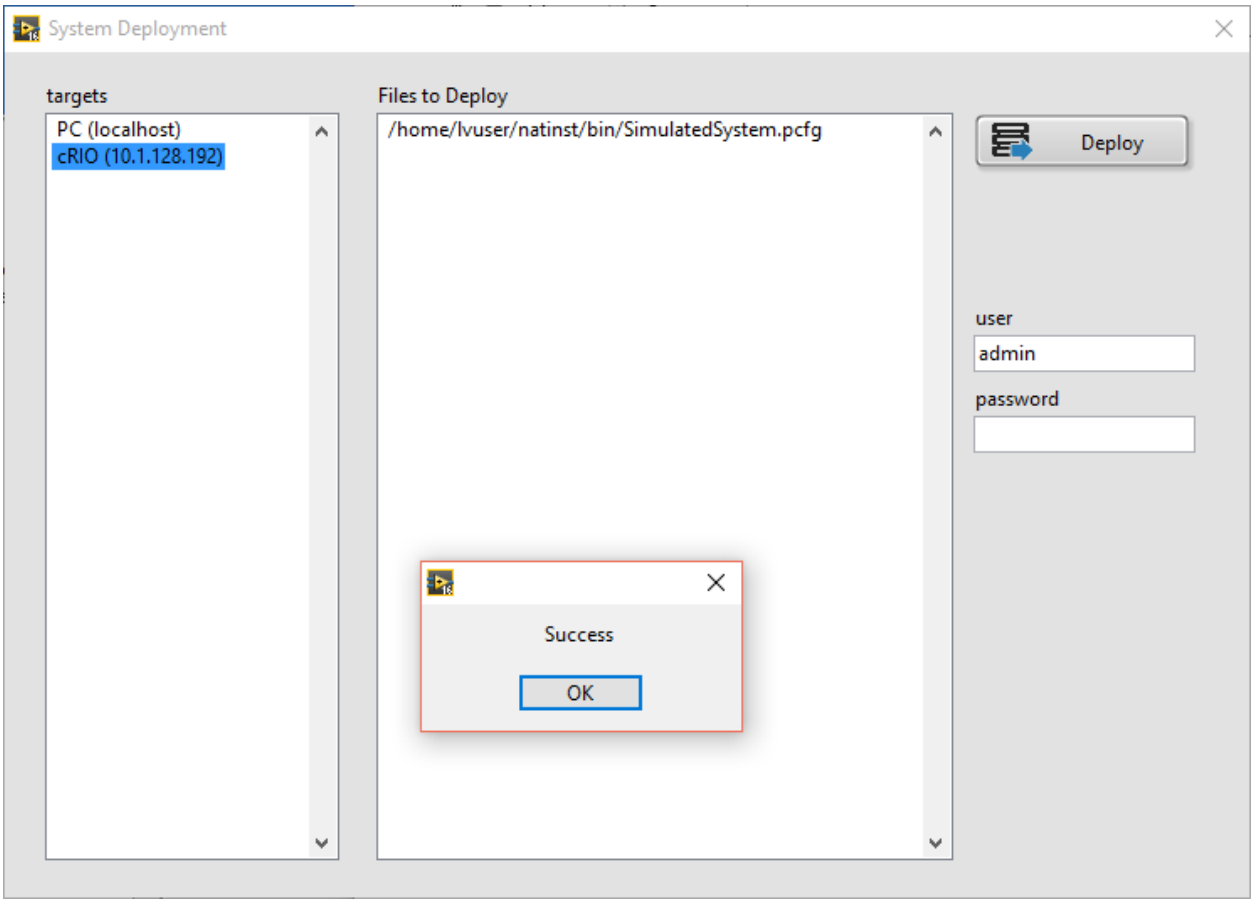
6. <WINDOWS ONLY> If running this configuration purely on your local Windows machine, modify the configuration so that the Engine on the cRIO runs on the PC instead. To do this, click the **Standard Engine (Simulation)** node under the engine and drag it up to the PC.



7. Ensure proper IP addresses for both UDP modules.
    a. <WINDOWS ONLY> For the UDP module under each engine, set the **send to address** field on the **Module Settings** tab for both UDP modules to **localhost**.
    b. <WINDOWS and cRIO> For the UDP module under each engine, set the **send to address** on the **Module Settings** tab on the PC UDP module to the IP Address of the cRIO and vice versa for the cRIO UDP module. Be sure to specify the actual IP address and not use a value of **localhost**.

The configuration is now ready to be loaded on your target(s) and executed. The local PC already has access to the configuration file, but in order for the cRIO to execute the configuration the file must be stored on the cRIO's hard drive.

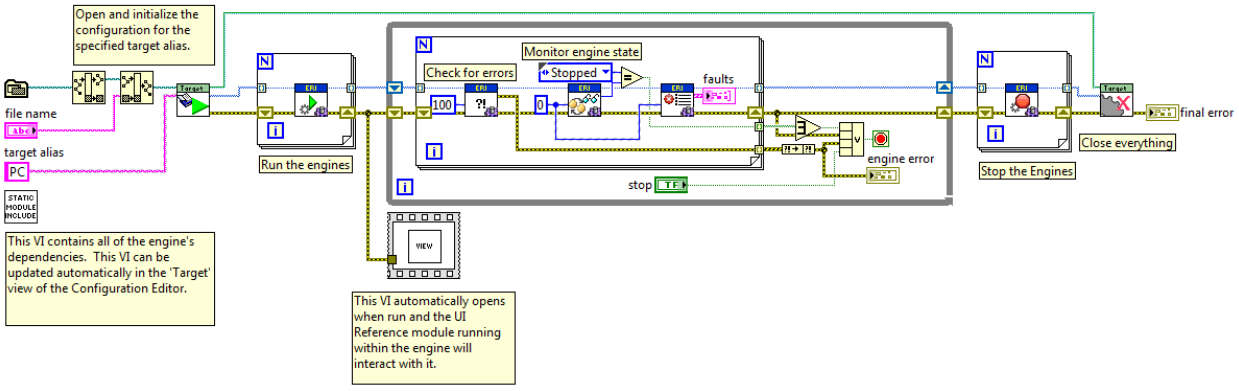
8. Save your changes to the configuration file by navigating within the editor to **File>>Save**.
9. <WINDOWS and cRIO> Click the cRIO target in the configuration editor and enter its IP address in the **IP** field. Then navigate to **Tools>>Deploy Tool**, click the cRIO target in the targets list, and then click the **Deploy** button. Ensure that you see a Success dialog appear as shown below.



'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. Leave the configuration editor open as you may want to refer to it again.

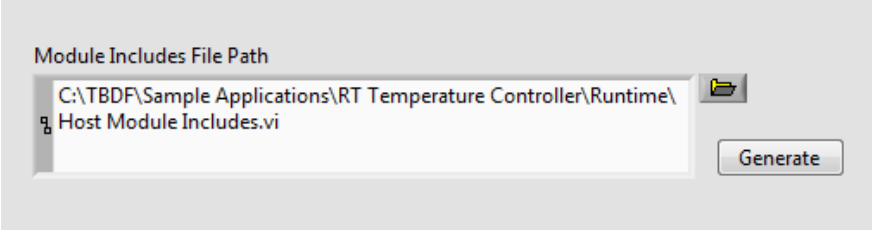
1. Open the **<LabVIEW Examples>\DCAF Examples\RT Temperature Controller\Runtime\Temperature Controller Example.lvproj** if it isn't already open. Then open the **Host Main.vi** from the project and inspect its Block Diagram.



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' DCAF module is able to write and read to controls and indicators directly as specified by the module's configuration. The 'UI Reference' DCAF module just needs the name of the VI in memory and the name of 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, DCAF modules, and any other plug-ins specified by the configuration file. This VI should be updated anytime a DCAF module is deleted or a new DCAF module is added to the configuration in order to reflect the new dependencies. Because the editor is aware of these dependencies, the editor can update this VI for you automatically.

2. <OPTIONAL> Navigate to the PC target in the editor and add the file path to the **Host Module Includes.vi** in the **Module Includes File Path** field.



While the editor can update these includes for you, you will still need to remember to click Generate when the plug-in dependencies change. Failing to do so will result in either a loading error or in having more dependencies than necessary in memory.

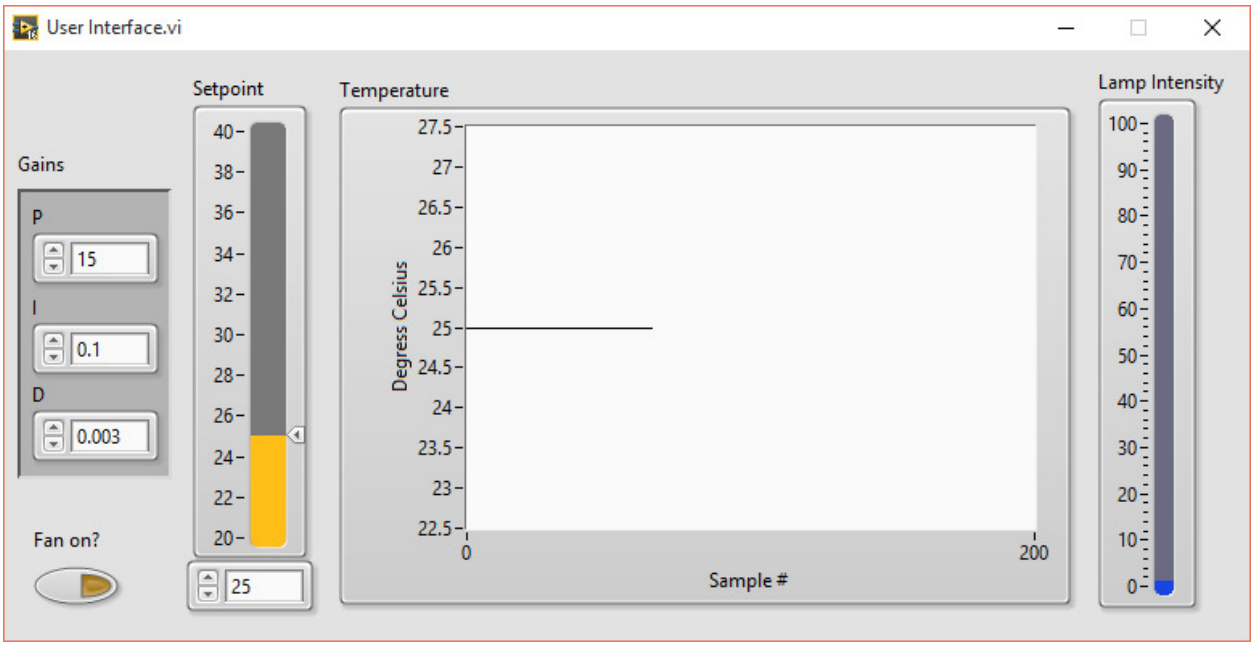
The framework was also designed so that DCAF modules could be built and deployed as plug-ins on disk which get loaded into memory dynamically at runtime instead of being statically included. When taking this approach the 'Host Module Includes.vi' would no longer be necessary, but source distributions would need to be created for each plug-in and code would need to be added to load those plug-ins into memory before loading up a configuration file.

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

We are now ready to run the host application.

3. Run 'Host Main.vi'.

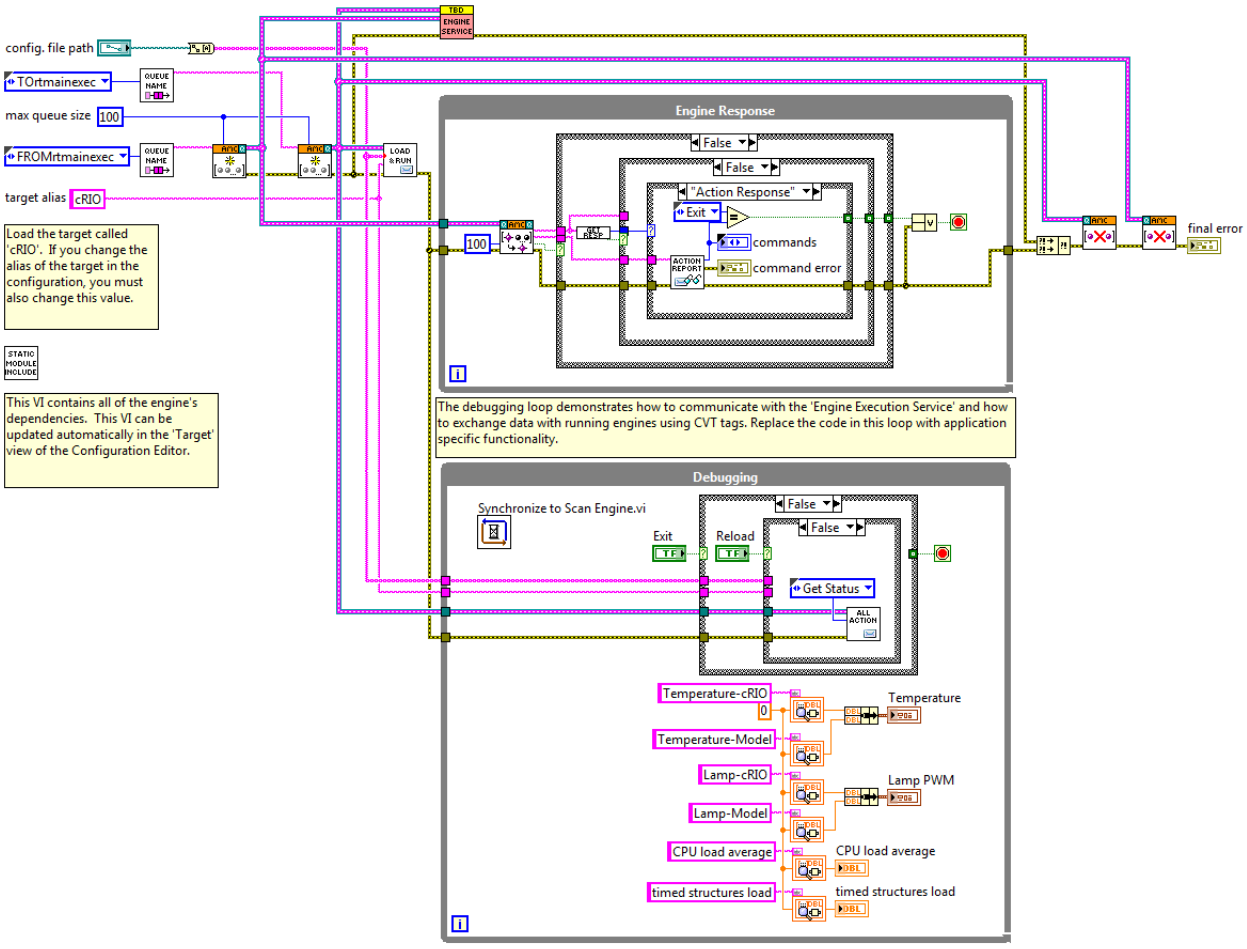
You should see the following User Interface appear which will allow you to specify the temperature setpoint, tune the PID gains, turn the disturbance on and off, and monitor the resulting temperature and lamp intensity. The controller only has a heating element so it will be unable to reach a setpoint below the ambient temperature (25 degrees by default). If you are running this example purely on the PC, the host will also run the Simulation Engine and you will also see data appear in the graph. Otherwise the UI will appear as below until data gets populated from the 'cRIO Main.vi'.



'cRIO Main.vi'

Whether you have access to a cRIO or not, let's open up and examine the cRIO application. Leave both the configuration editor and the 'Host Main.vi' open and running as they will still be referred to.

1. Open up and inspect 'cRIO Main.vi' in the project.



This application is 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'.

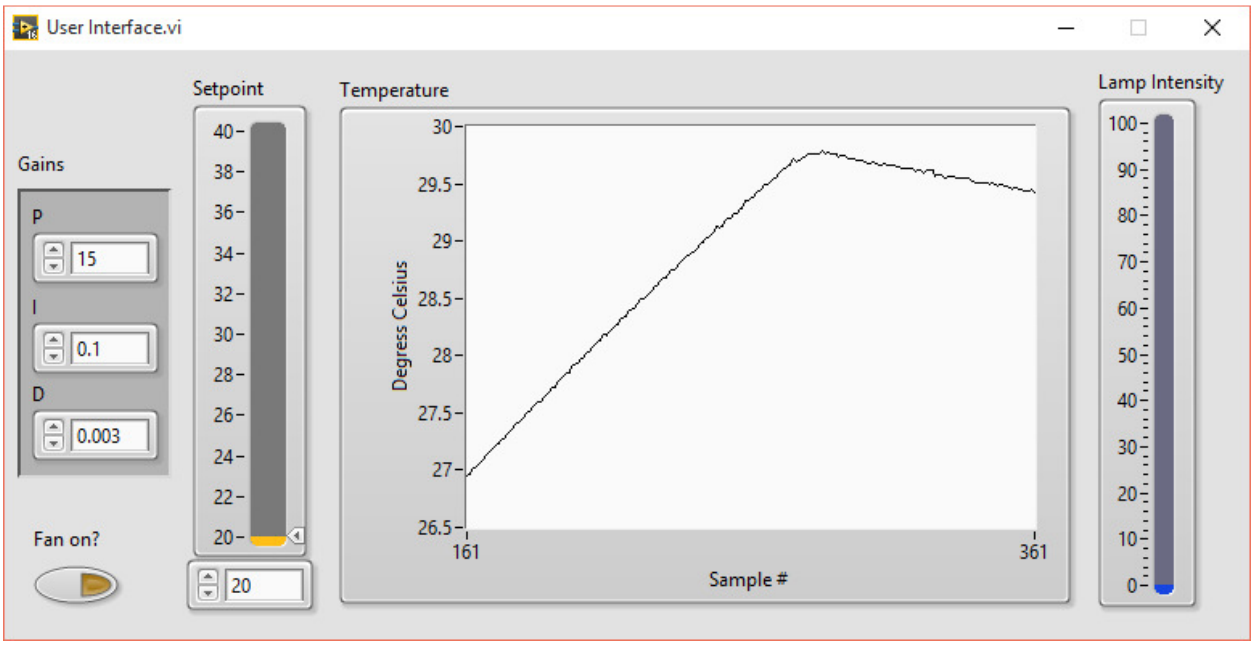
2. <OPTIONAL> Navigate to the cRIO target in the editor and add the file path to the **cRIO Module Includes.vi** in the **Module Includes File Path** field.

There is also a debugging loop with a collection of application specific Current Value Table (CVT) tags. Ignore this code for now. It isn't used for this example and will be explored as part of the 'FullSystem.pcfg' file which we will examine later.

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'.

3. <WINDOWS and cRIO> Change the cRIO target's IP address in the project to match the IP of the controller you want to use.
4. <WINDOWS and cRIO> Run 'cRIO Main.vi'.

Once running, you should now see data appear on the 'Host Main.vi'. (If data doesn't appear, 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.)

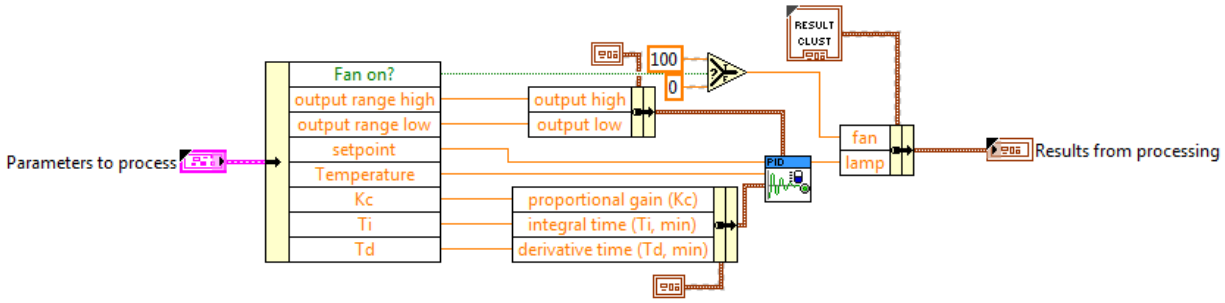


User Control DCAF Modules

Now let's take a second look at the four DCAF modules used for this example within the configuration editor: Temperature Controller Logic, Temperature Controller Model, UI Reference, and UDP. Of these, the UI Reference and UDP modules could be valuable tools for any project (which is why they are installed to vi.lib). However, the Temperature Controller Logic and Temperature Controller Model are both specific to this example application.

This is a common theme for many applications. Data services and I/O plug-ins tend to be functionality that is likely usable on other projects, while at least some aspects of control logic tend to be application specific. In addition to many of the pre-built data service and I/O DCAF modules, the DCAF framework also includes a User Control Module sample project to aid users in the creation of their own control logic plug-ins. This template can be used for any functionality with a fixed set of inputs and outputs (UDP and UI Reference both support a dynamic number of inputs) and was used to create the Temperature Controller Logic and Temperature Controller Model functions.

This template can make the creation of new plug-ins as simple as putting control logic within a single VI. To examine the single VI implemented for the Temperature Controller Logic module, open up **<LabVIEW examples>\DCAF Examples\RT Temperature Controller\Modules\Temperature Controller Logic\ module\execution\User Process.vi.**



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 User Control Module script from LabVIEW **Tools>>DCAF>>Launch User Control Module Scripting Utility…** The script uses the information in these clusters to regenerate a module including its editor UI. In the simplest case, creating a DCAF module to execute code inline with the engine can be as simple as using the User Control Module Sample Project and placing your logic within a single VI. For step by step instructions on creating a new User Control Module, see the 'Building a New User Control Module' section at the end of the document.

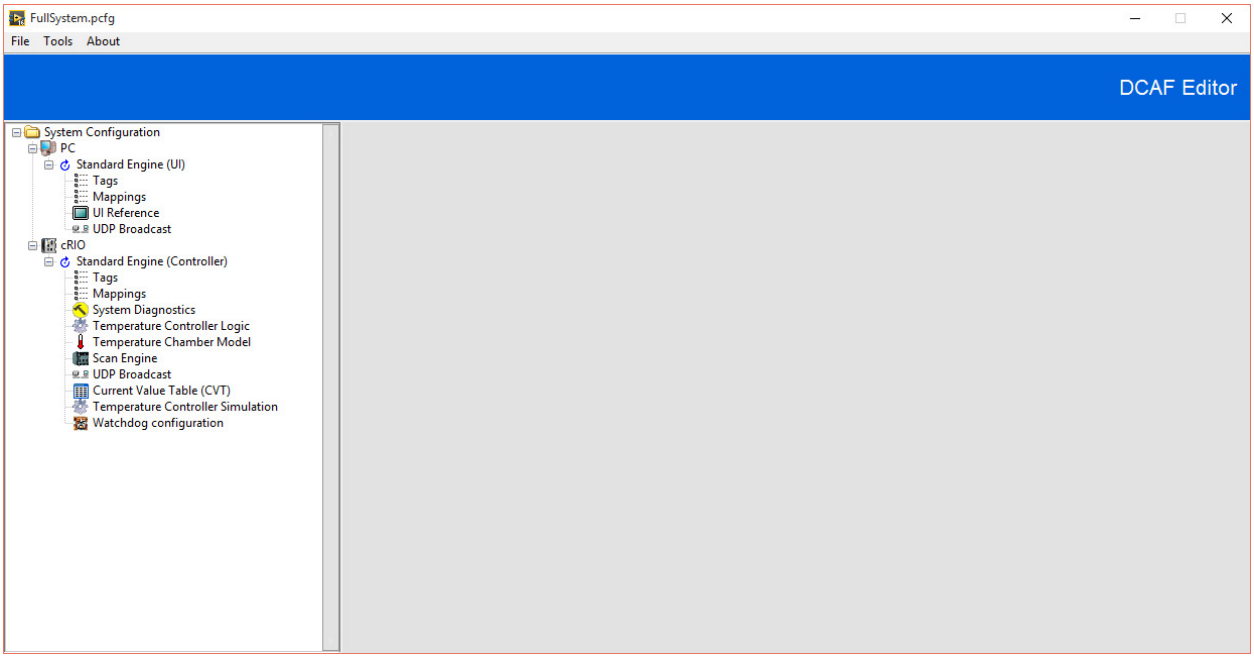
Simulated System Example Summary

The Simulated System example demonstrates how to create and modify a system configuration in the configuration editor, deploy a configuration to a target, and update that target's plug-in dependencies. We have also discussed the Sample Project behind the creation of the example's control logic.

Full System Example

This example makes use of real-world I/O connected to a temperature chamber. Although you likely don't have the hardware dependencies necessary to actually run the example, much can still be learned from examining the implementation.

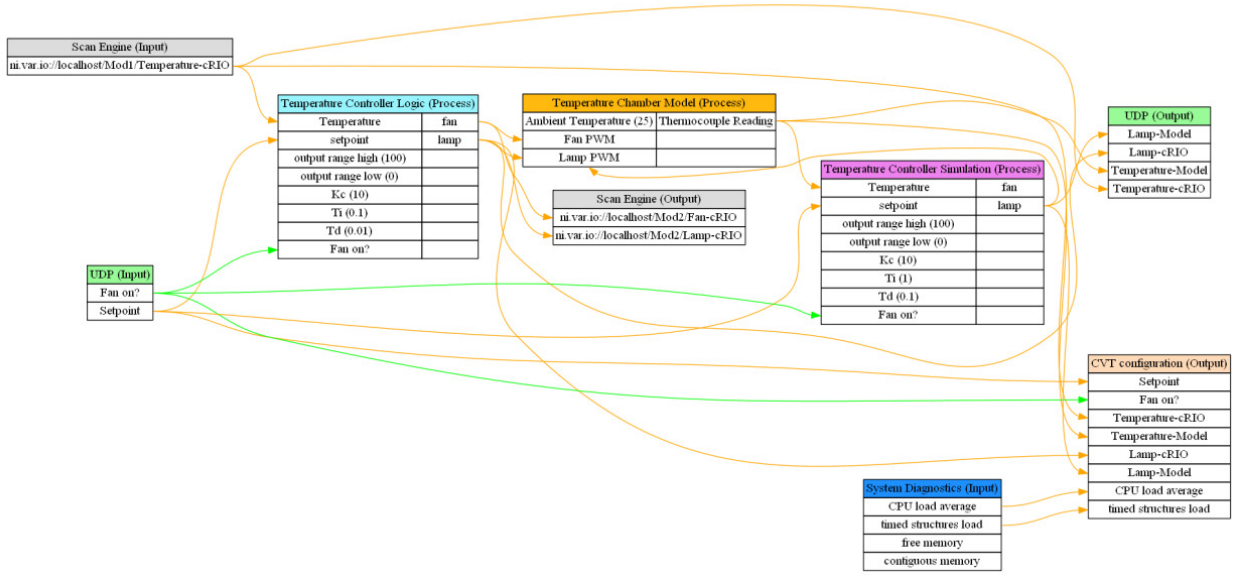
1. From the Standard Configuration Editor, open the **FullSystem.pcfg**.



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

There are also some DCAF modules 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 DCAF module is the Current Value Table (CVT) which is used to expose specific data in the engine for global 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 a visual representation of the DCAF modules, Channels, and Tags for this configuration. The diagram uses the same conventions as the similar diagram above, but omits the Tag labels for the wires.



Because the 'FullSystem' configuration refers to additional modules (relative to the 'SimulatedSystem'), the following additional steps are required before you run the 'FullSystem' example:

1. Use MAX, the project, and DCAF to configure your I/O and remap channels as necessary
2. Use MAX to add NI-Watchdog to your cRIO target software
3. Use *DCAF Configuration Editor>>Tools>>Script Include VI* to rescript the cRIO Module Includes VI
4. Use *DCAF Configuration Editor>>Tools>>Deploy Tool* to deploy 'FullSystem' configuration
5. Set **file name** in Host Main VI and **config. file path** in cRIO Main VI to refer to FullSystem.pcfg

Full System Example Summary

There are a few key takeaways from this example. The first is that the example uses the exact same DCAF modules, 'Host Main.vi', 'User Interface.vi', and 'cRIO Main.vi' as the Simulated System example. The only difference between the two examples is the additional DCAF modules, Tags, and Mappings added to the cRIO target in the configuration file and the additional dependencies in the cRIO target's Module Includes vi.

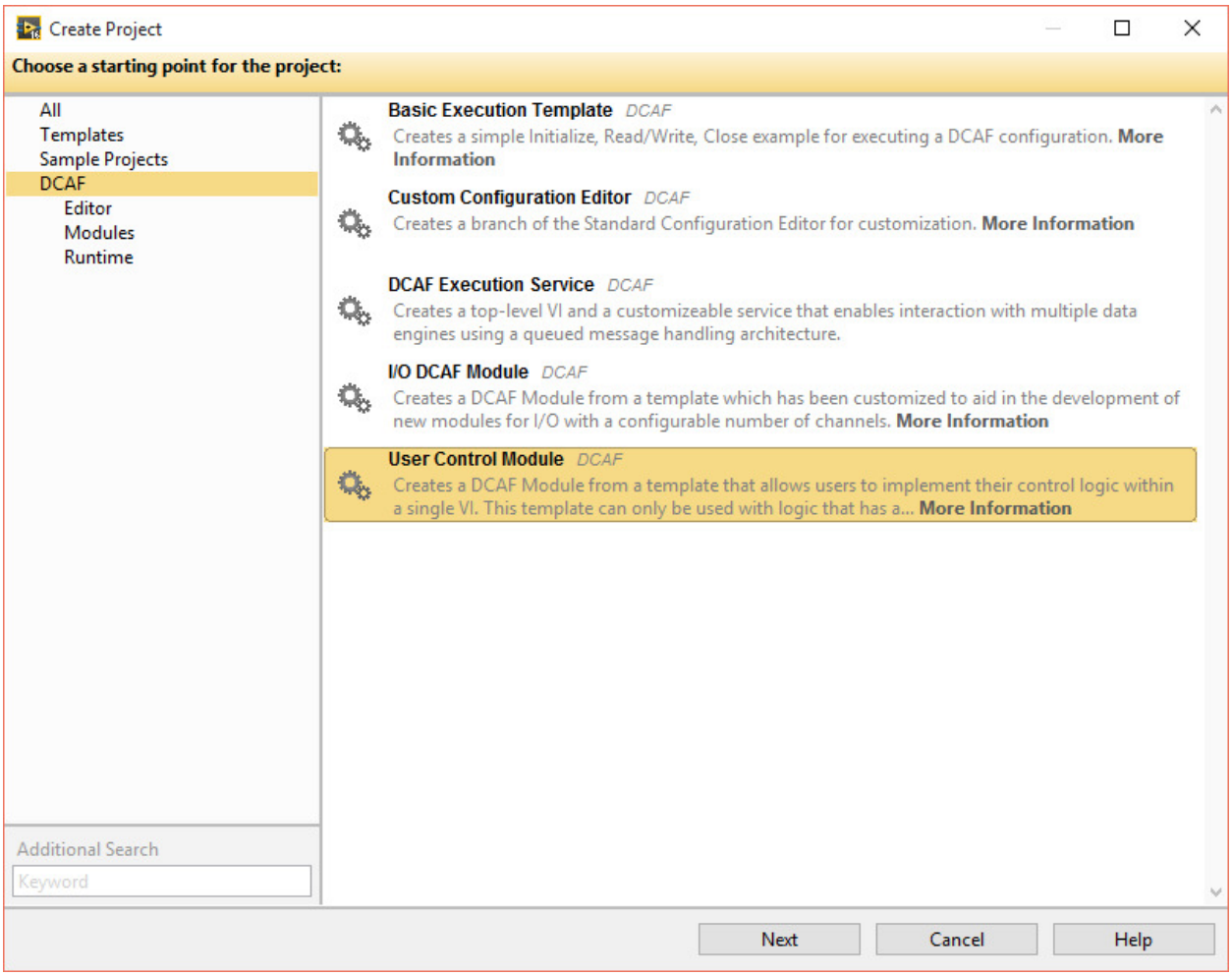
This example also demonstrates how the framework can allow users to easily switch between simulated I/O and real world I/O for the benefits of testing and productivity. The framework also allows the execution of any simulated I/O alongside the real I/O for the purposes of comparison and diagnostics.

Another takeaway is that despite the benefits of running code within the framework through the creation of new DCAF modules, various escape hatches like the CVT module allow for the development of code that can easily execute alongside the framework. Using mechanisms like the CVT, the framework can be used to integrate single-point data from a variety of sources and expose it through a common API.

Building Your Own Application

We have now covered how the configuration editor, the Host Main, the cRIO Main, and the User Control Module template can be put together with existing DCAF modules to implement a simple application. DCAF includes a collection of sample projects to help users quickly create a majority of the code required for their application.

To find these sample projects, navigate in LabVIEW to **File>>Create Project…** and click on the DCAF category.



The **Basic Execution Template** sample project can be used to create a new VI that is very similar to the 'Host Main.vi'. This sample project is recommended for simple applications where getting something up and running quickly and easily is desired.

The **DCAF Execution Service** sample project can be used to create a new VI very similar to the 'cRIO Main.vi'. This sample project is a good starting point for applications that need the ability to load new configurations without rebooting or need to receive engine commands from another target.

The **User Control Module** sample project can be used to create new DCAF modules that are similar to the Temperature Controller Logic DCAF module. This sample project provides a table where a user can specify all of their DCAF module's inputs and outputs as well as a few other properties. Once entered, it will then script out the clusters and other code necessary so that users can put their control logic in clearly defined locations.

The **Custom Configuration Editor** sample project can be used to create your own copy of the Standard Configuration Editor for customization. The use of this sample project is not recommended unless significant customization of the configuration editor is required. While much of the Standard Configuration Editor's code is built from underlying framework elements that are updateable, it also includes some of its own features. The use of this sample project amounts to a branch from the Standard Configuration Editor that will have to be maintained independently.

Finally, the **I/O DCAF Module** sample project can be used to develop modules similar to the Scan Engine DCAF module. This sample project only scripts out a minimum set of functionality and requires more detailed knowledge of the framework to be used successfully.