Distributed Control and Automation Framework Hands-On Session

This hands-on covers the basics of implementing an application in the Distributed Control and Automation Framework (DCAF), 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 the [Reference Designs for Distributed Control Systems](https://decibel.ni.com/content/projects/reference-designs-for-distributed-control-systems) portal on [ni.com/referencedesigns](https://decibel.ni.com/content/groups/reference-designs).

For more detail on the concepts used by the framework, consider attending session AD7049 (Tools for Designing Distributed Control Applications) at 11:45 AM in 16B. For a more in-depth look at the framework itself, attend TS6859 (Tag Bus Data Framework for Embedded Control Applications) at 3:30 PM on 8/5/15 in room 14.

## Set-up (Already configured on Alliance Day machines)

Navigate to <https://decibel.ni.com/content/docs/DOC-41727> and download the .vipc file attached to that page. This package can be installed to any LabVIEW version from 2013 to present.

# Exercise 1: Configuring a Simple Temperature Control Application

In this exercise you will see how to configure and run an existing application using existing modules.

Open the temperature controller project by navigating to **Help >> Find Examples…** and then searching for “TBD”. You can also navigate to <LabVIEW 2015>\examples\TBD Examples\RT Temperature Controller\Runtime.

Under the My Computer target, you’ll see Temperature Chamber Quick Start Guide.pdf. Open this PDF and follow the instructions for the <Windows Only> track. Note: Do not follow “<Windows and cRIO>” instructions.

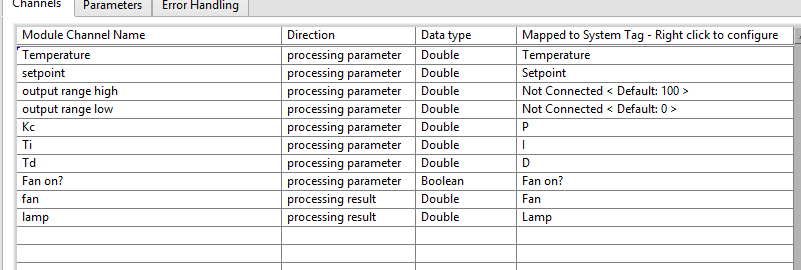
# Exercise 2: Creating a Custom Module

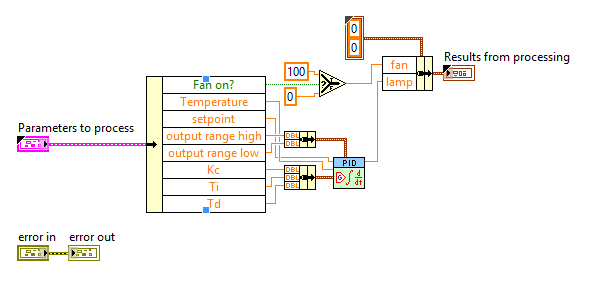
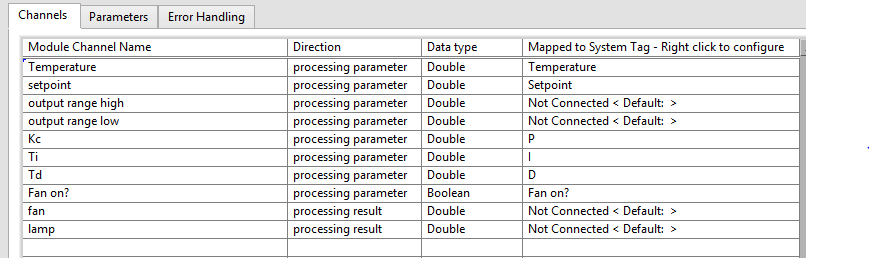
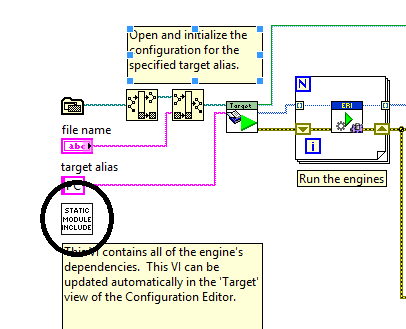
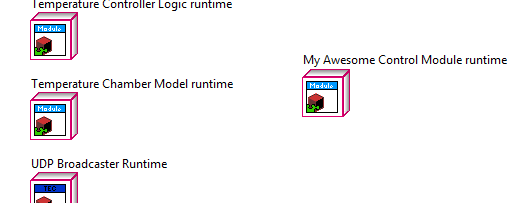
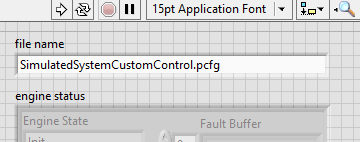
In this exercise we will implement a custom control algorithm for our temperature controller. This can be actually custom, or a simple PID controller as is used by the original example.

1. Navigate to **Project >> Create Project…**
2. In the tree on the left, select **TBDF >> Modules**.
3. Select **User Control Module**.
4. Enter a new name for your module and select an appropriate project path.
5. Add the following parameters to your new module as Channels. 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 |

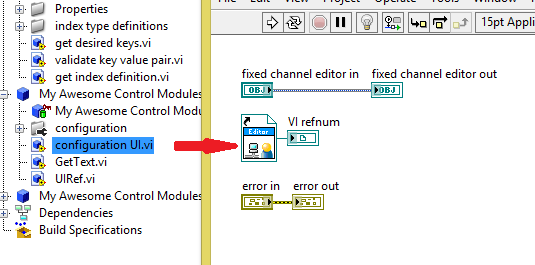
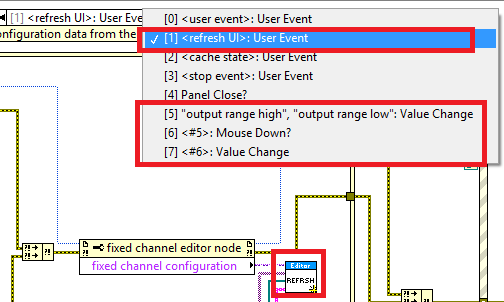
At the end of this process you should have a channel configuration which looks like this:



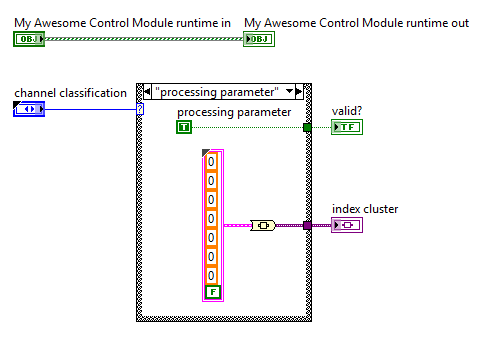
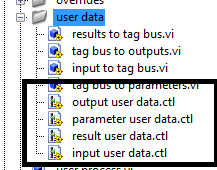
1. Press **Finish**.
2. Your new project will appear. Navigate to *YourModuleName* Runtime.lvclass and open *user process.vi*. This method should have two clusters, one input and one output, which match the list of tags above.
3. Open *process.vi* in the overrides folder. On either side, scripted accessors 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 do the same thing for the configuration class (*YourModuleName* Configuration.lvclass).
   3. Because you used the script, you can leave most options as the default and press **Run**.
4. Return to *process.vi*. *User process.vi* is the function being called in the middle and is the function you will implement. Add any control code you desire to this module. If you want to match the original module you just used, you can add the following:
   1. Drop down an instance of PID.vi from **Control and Simulation >> PID >> PID.vi**.
   2. Unbundle *Parameters to process.Temperature* and wire as the process value
   3. Wire *Parameters to process.setpoint* as the setpoint
   4. Bundle *Parameters to process.output range high and Parameters* to *process.output range low* and wire the cluster to output range
   5. Bundle *Parameters to process.Kc, Ti, and Td* and wire the cluster to PID gains
   6. Wire *Parameters to process.Fan on?* to a select function or case structure and set the two values as T=100 and F=0. Wire the output of this function to the *Results from processing.fan* value
   7. Wire the PID output to *Results from processing.lamp*.
   8. The result should look something like this:  
      
5. Save the new project and close it.
6. Reopen *Temperature Controller Example.lvproj* if you closed it, and reload the standard configuration editor (Open **Tools >> DCAF >> Launch Standard Configuration Editor…**).
7. Navigate to **Tools >> Edit Plugin Search Paths**.
8. Press **Add** and navigate to the location of your new control module.
9. Reopen <LabVIEW 2015>\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.
10. Now, **right click** on *Standard Engine (Simulation)* and select **Add >> Other >> *YourControllerModule***. Then, select this new module from the tree.
11. 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 this:  
    
12. Now, right 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.  
    
13. 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.**
14. Return to your module’s configuration screen and map channel “fan” to tag “Fan” and channel “lamp” to tag “Lamp”.
15. Select to **File >> Save.**
16. Open Host Main.vi (our top-level application VI) and navigate to Host Module Includes.vi on the diagram. 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.  
    
17. Drag an instance of *YourModuleName* Runtime.lvclass from the project onto the diagram of Host Module Includes.vi. This ensures that your new module is always loaded into memory.  
    
18. Now save and close this VI and return to Host Main.vi. On the front panel, enter the name of your configuration file (we suggested SimulatedSystemCustomControl.pcfg) in the *file name* control.  
    
19. Press **run**.
20. The behavior should match that of the original controller if you used the same code.   
    Debugging Note: While *process.vi* is a shared reentrant method and is therefore difficult to debug, *user process.vi* defaults to being non-reentrant and debuggable.

# Exercise 3: Creating a Custom Configuration Editor UI

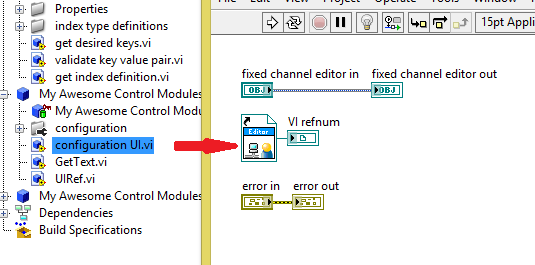
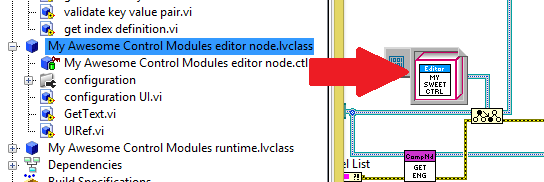
In this exercise, you will implement a custom user interface for configuring your controller module. This will likely take more time than we have available.

1. Close all projects except for the project you just scripted for your module.
2. To add a brand new editor you could follow the steps in Appendix B. This is not required, as we have provided a starting point.
3. To use the provided starting point, drag the contents of *<Desktop>/lesson three starting point* into *YourModuleName* editor node.lvclass.
4. Define *HandsOnCfgUI.vi* as your module’s editor.
   1. Right click on your editor node class and select **New >> VI for Override…**
   2. Select UI ref.vi and press **OK**.
   3. Remove the call parent method node, drop down a new **Static VI Reference** and drop your new *HandsOnCfgUI.vi* method into that static reference.  
      
5. Open *HandsOnCfgUI.vi* and look at the front panel. It is expected that this VI is broken right now. The controls provided are named identically to the channels in your module. This is for convenience. As you saw in the original, more complex UI, it is definitely possible to make the front panel more dynamic. Because time is limited, we’ve made the assumption that the editor node must be updated to match the channels in the configuration.
6. Fundamental to this particular implementation is that the combo boxes on the front panel map directly to channels in your module. That is, the combo box with the label “Temperature” represents data associated with the channel “Temperature”. In this case the data being displayed is what tag is mapped to that channel.
7. Go to the block diagram and look around. You will be implementing the code for the other VIs provided, which is called in the cases below. Each block diagram has instructions for what is required along with a list of all methods needed. These methods are all available on the palette, but we’ve picked out the appropriate ones to save you time. You’ll also see that much of the infrastructure (registering for the right events, locking and unlocking DVRs, etc) has been completed for you. The focus is just on using the DCAF APIs.   
   
8. If you run into any issues, ask for help or take a look at the solution.
9. Once you’ve completed making your changes, load up the standard configuration editor and try to use your new UI. Assuming you completed step 4 correctly, it should at least load correctly.   
   Debug Note: Once loaded, you can right click on the front panel (as shown in the subpanel of the main editor) and select “Open Block Diagram”). This makes it easy to debug your code as its running. If you need to make changes, you don’t need to stop the entire editor, you can simply click away from your module and this will unreserved your UI, allowing you to make edits.

# 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.  
   

# Appendix B: Creating a new editor UI

1. Right click on *YourModuleName* editor node.lvclass and select **Go To Parent Class**.
2. Open *configuration UI.vi* in the parent class (*control module editor.lvclass*) and save a copy into your editor class (save a copy into the project, then drag and drop into your editor). Be sure to name it something else, like *MyCfgUI.vi.*
3. Right click on your editor node class and select **New >> VI for Override…**
4. Select UI ref.vi and press **OK**.
5. Remove the call parent method node, drop down a new **Static VI Reference** and drop your new *MyCfgUI.vi* method into that static reference.  
   
6. Open your instance of *MyCfgUI.vi*, press **ctrl+m** and then **unlock**. Then, drag your editor class into the data value reference. This makes sure your new UI references your class rather than the parent class.  
   
7. The front panel shown is exactly what you have been working with so far. The event structure and surrounding code already have everything needed to cooperate with the framework. Specifically, be sure that you don’t remove the following event cases:
   1. [0] <user event>: User Event
   2. [1] <refresh UI>: User Event
   3. [2] <cache state>: User Event
   4. [3] <stop event>: User Event
   5. [4] Panel Close?