Getting Started with the Tag Bus Data Framework

Contents

[Introduction 1](#_Toc394668929)

[Requirements for the Getting Started Guide Projects 2](#_Toc394668930)

[Installation 2](#_Toc394668931)

[Creating a Hello World Application 2](#_Toc394668932)

[Creating a Single Engine RT Project 2](#_Toc394668933)

[Creating a Configuration Editor 5](#_Toc394668934)

[Running the System 10](#_Toc394668935)

[Developing a New User Code Module 10](#_Toc394668936)

[Introduction to Development 11](#_Toc394668937)

[Implementing Function-Oriented Code 11](#_Toc394668938)

[Implementing State-Oriented Code 17](#_Toc394668939)

[Testing the Module 21](#_Toc394668940)

[Developing a Configuration User-Interface 23](#_Toc394668941)

[Generating a Configuration in the Editor 27](#_Toc394668942)

[Developing a New I/O Module 28](#_Toc394668943)

[Advanced Topics 28](#_Toc394668944)

[Creating a Hard-Coded RT Application 28](#_Toc394668945)

# Introduction

The Tag Bus Data Framework is a series of tools used to develop embedded control applications in LabVIEW. The framework is configuration- and plugin-based, making it easy to rapidly create and adjust applications to suit changing requirements. In addition, it is intended to be as reliable as possible while maintaining determinism and providing error-handling and error-reporting capabilities.

This getting started guide covers the use of the standard engine, the configuration editor, and developing a new plugin.

# Requirements for the Getting Started Guide Projects

Be sure you have the core package set, along with the packages for the variable engine runtime, variable engine editor, CVT runtime, and CVT editor.

In addition, this will require a scan-engine-capable chassis with at least one input module and one output module.

# Installation

The Tag Bus Data Framework (TBDF) consists of a set of core libraries and an associated set of plugins.

At the core is the Tag Bus which provides the data transfer mechanism between all pieces of the framework.

On top of this tag bus is the Tag Bus Module Framework which contains the interfaces for module runtimes, module configurations, and the concept of a module channel (which is a type of tag).

The Tag Bus Execution Engine Interface provides the interface for managing an engine, as well as the configuration APIs necessary for generating tags and mapping channels to tags.

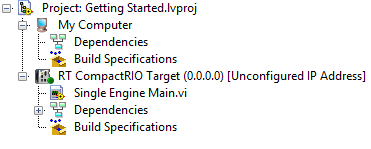
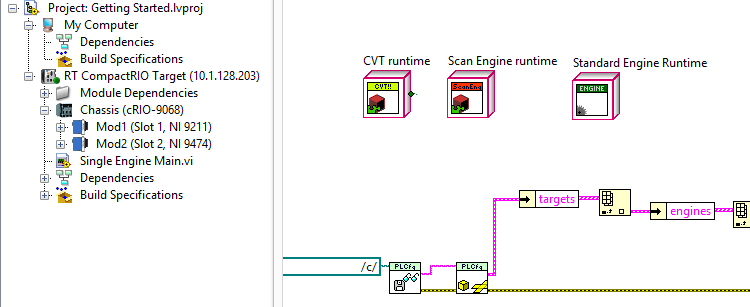
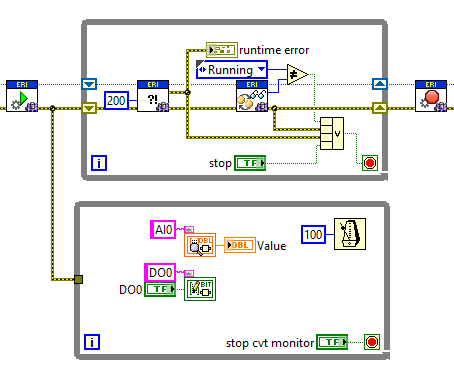
Finally, the Tag Bus Data Framework includes sample projects, templates, and the other pieces necessary for generating a fully-functioning system like file loaders, class locators, and so on.

In addition to these pieces, which are installed to <vi.lib>/NI, there are a number of plugins which are installed to <vi.lib>/NI/TBM Modules and …/TBM Engines. At present, the only extant engine is the “Standard Engine” which is included in the core VI Package Configuration (VIPC). There are numerous modules. Typically, plugins will come in two separate packages. The first piece (typically called “… Tag Bus Module”) is the runtime, while the second (typically “… Tag Bus Module Editor”) is used to plug into the configuration editor. This allows for a cut down install if only the runtime is required. Runtimes depend on the tag bus module framework or, in the case of engine runtimes, the tag bus execution engine interface. Editors depend on the tag bus data framework package.

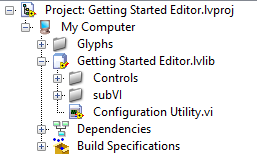
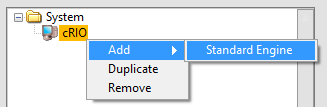
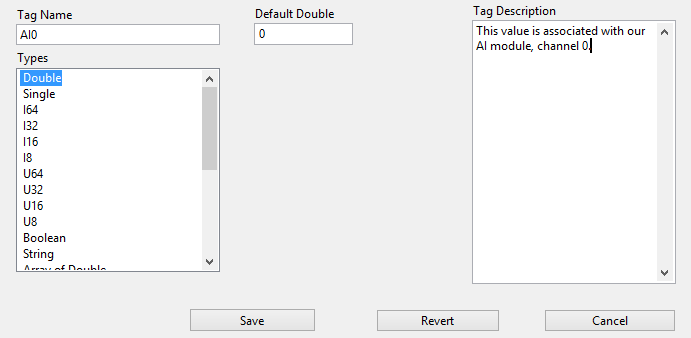
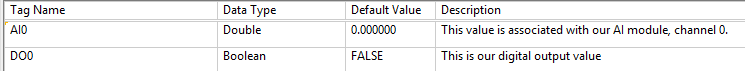
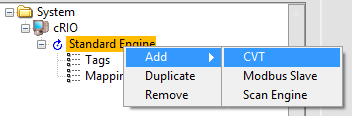
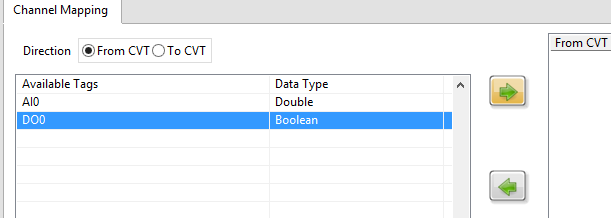
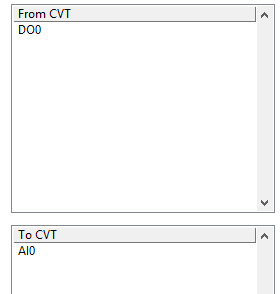
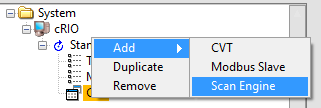
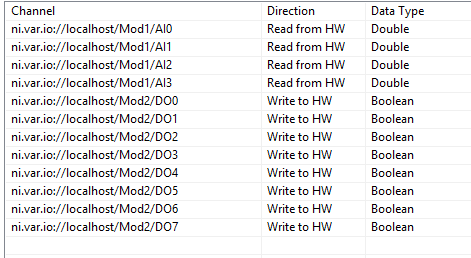
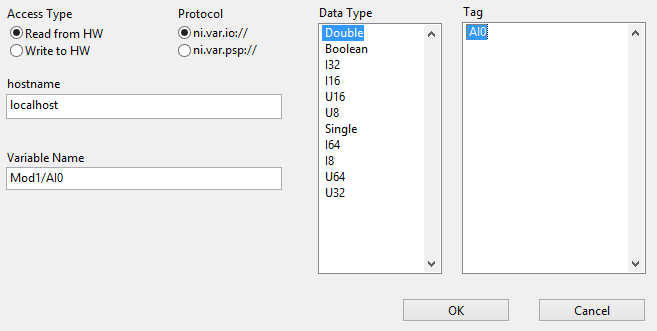
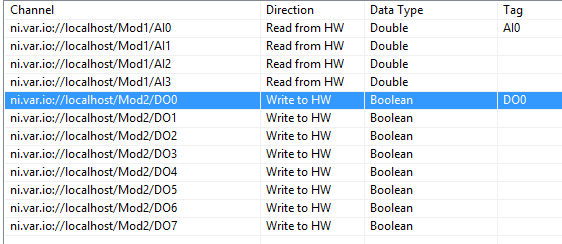
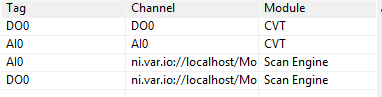
# Creating a Hello World Application

The TBDF ships with two sample projects for developing a new RT application. One is a simple single-engine single-thread application which will meet most common use cases. The other is a much more complex and full-featured queued message handler, which exposes all framework features but in turn requires additional development effort from users. This guide will cover the simpler single-engine project.

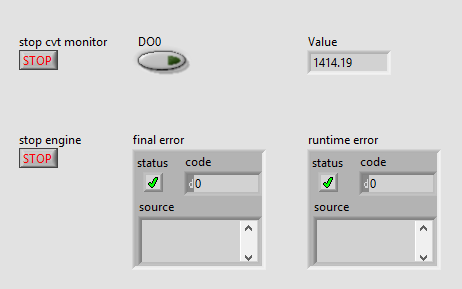
## Creating a Single Engine RT Project

1. From the LabVIEW getting started screen, press the **Create Project** button.
2. From the dialog, select **Sample Projects** from the tree on the left hand side.
3. On the right, *Single Engine System* should appear near the top of the list of available sample projects. Select it and press **Next**.
4. Fill in the fields for project name and location, and press finish. The resultant project should appear shortly.  
   
5. Right click on the RT CompactRIO target and select **Properties**. On the *General* tab, enter the IP address of your target and press **OK**.
6. Right click on your cRIO target and select **New >> Targets and Devices…**, and add the appropriate chassis for your target.
7. For this getting-started guide, ensure the target is in scan mode by navigating to chassis properties and selecting the scan interface programming mode. Then right click on the chassis and select **New >> C Series Modules…** and add any modules you have in the system. Finally, deploy chassis settings by right clicking on the chassis and selecting **Deploy All**. You may choose to use FPGA or Hybrid programming in your application.
8. Open Single Engine Main.vi and investigate the code. From left to right, the code is doing the following:
   1. Loading a configuration file (.pcfg) from a specified path. This path must be provided by the developer.
   2. Parse the configuration file to generate a *system configuration*.
   3. Retrieve the list of engines from the system configuration file – this project assumes a single target and a single engine.
   4. Load the modules, configurations, and engines and initialize the engine.
   5. Begin running the engine.
   6. Monitor the engine for error messages and for state. If the engine gets an error from any module or enters a non-running state, the loop will exit and the system will uninitialize.
9. For the purposes of the getting started guide, we will be using the scan engine and CVT. To ensure that LabVIEW loads the classes into memory when deploying to the target, drop an instance of each module’s runtime onto the block diagram of Single Engine Main.vi
   1. Navigate to *vi.lib\NI\TBM Modules\CVT Access\module\execution* and drag CVT runtime.lvclass onto the block diagram.
   2. Navigate to *vi.lib\NI\TBM Modules\variable\module\execution* and drag Scan Engine runtime.lvclass onto the block diagram.
   3. Navigate to *vi.lib\NI\* *TBM Engines\Standard Engine\Runtime* and drag Standard Engine Runtime.lvclass onto the block diagram.
10. After step 9, the project should look something like this:   
      
    Here, you can see that we pulled the module-related dependencies into the project. This is not required.
11. Optional: To develop a real-time executable, follow these steps:
    1. Right click on Build Specifications and select **New >> Real-Time Application**.
    2. Under **Source Files**, add *Single Engine Main.vi* to Startup Vis
    3. Under **Additional Exclusions**, uncheck every box. This ensures that all required files are included in the executable.
    4. Press **OK**.
12. In *Single Engine Main.vi*, enter a path for your configuration file. For now, enter something like */c/test.pcfg* or *c:/test.pcfg*, depending on the OS used by your cRIO target.
13. Create a second, parallel loop in *Single Engine Main.vi*
    1. Create a while loop and wire the error wire from *Start.vi* to the left side of this loop.
    2. Inside the new loop, add a wait timer and select an appropriate period like 100 ms.
    3. Identify the modules available in your development system. In the test system used by this guide, there are four AI (AI0-AI3) and 8 DO (DO0-DO7).
    4. Drop down a CVT read function (Data Communication >> Current Value Table) and change the polymorphic type to a type that matches your input module. In this case, the AI modules provide *double* values.
    5. Wire up a string to the *Tag Name* field and enter the name you wish to use in the CVT. For simplicity, we will use “AI0”.
    6. Create an indicator on the *Value* output and change the terminal label to the name of your tag. In this case, we use “AI0”.
    7. Drop down a CVT write function and change the polymorphic type to a type which matches your output module. In this case, DO modules require *boolean* values.
    8. Wire up a string to the *Tag Name* field and enter the name you wish to use in the CVT. For simplicity, we use “DO0”.
    9. Create a control on the *Value* input and change the label to the name of your tag. In this case, we use “DO0”.
    10. Note down the names and types of the tags selected.
    11. The final result should be similar to this diagram:  
        
14. Save and close the project

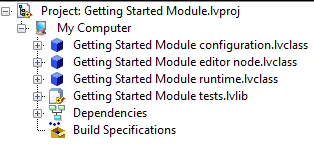
## Creating a Configuration Editor

1. From the LabVIEW getting started screen, press the **Create Project** button.
2. From the dialog, select **Templates** from the tree on the left hand side.
3. On the right, *Module Configuration Editor* should appear near the top of the list of available sample projects. Select it and press **Next**.
4. Fill in the fields for project name, library name, and location, then press finish. The resultant project should appear shortly. This project functions well as-is, but can be modified to suit the needs of the project. Everything in the generated lvlib and glyphs folder may be modified by the user.  
   
5. Open *Configuration Utility.vi* inside of the generated library and press run. The editor should automatically discover all modules located in vi.lib/TBM Modules and vi.lib/TBM Engines.  
   Note: The first run may take a little while classes are loaded for the first time.  
   Pitfall: If your editor is configured to load any modules which are actively in development, and that module is currently broken, the code can get into a state where the main editor loads but then is broken when it attempts to load your module in-development.
6. If necessary, navigate to **Tools >> Edit Plugin Search Paths** and add any additional search paths where you are storing your plugins.
7. To start configuring the system, right click on *System* in the tree view and add a *cRIO* target.  
   
8. Then, add a *Standard Engine* to the cRIO (or any other engine you have available in you system.  
   
9. Adding an engine gives you two nodes automatically: *Tags* and *Mappings*. *Tags* is a list of all tags in the engine. Engines have tags, while modules have channels. *Mappings* shows the link between channels and tags. Since we don’t have any modules yet, we won’t be able to see anything here.
10. Click *Tags* and then press the <**Add>** button. In the dialog configure a tag which matches the CVT tag name you selected in your runtime project. In our case, we will add a tag “AI0” as a *double* tag. Then, we’ll add “DO0” as a boolean.  
       
    After correctly configuring the tag list, we have a table like this:  
    
11. At this point, save by pressing Ctrl+S or going to **File >> Save**. Enter the name you selected in step 12 of the runtime project steps. This can be saved on disk anywhere, as we’ll need to deploy it to the RT target.
12. Add a CVT module to the standard engine by right clicking on the engine.  
    
13. Select the *CVT* view and configure it as follows:
    1. Select *From CVT* as the direction.
    2. Since DO0 is data moving *FROM* the CVT *TO* the Engine, select DO0 and press the *right arrow* next to the *From CVT* list.  
       
    3. Select *To CVT* as the direction.
    4. Select *AI0* and press the *right arrow* next to the *To CVT* list.  
       
14. Add a scan engine module to the system.  
    
15. Select the scan engine module in the tree. Here, we can either add channels manually or import a project. For simplicity, let’s import a project.
    1. Press **Import Project**.
    2. Navigate to the runtime project you generated earlier and load it.
    3. The configuration should load and provide a list of channels.  
       
    4. We don’t need anything except for AI0 and DO0. But, since channels and tags are separate, we can leave them here. To map these new channels to tags, select one. In our case, we select *ni.var.io://localhost/Mod1/AI0* and press the **settings** (*gear*) button.
    5. In the dialog, you should be able to see the configuration options along with any tags which can be mapped. In our case, we just have one: AI0. Select it, and press **OK**.  
       
    6. Do the same for your output channel – in our case, DO0. The final result will look like this:  
       
16. Navigate to the *Mappings* view. You should see a large number of channels, your two tags, and four mappings. It is better to change mappings from the views of individual modules, but they can be added or removed here if necessary. At present, the *mappings* view should look something like this:  
    
17. Save this final configuration by pressing Ctrl + S or navigating to **File >> Save**.
18. Close the editor.

## Running the System

1. Now that we have a configuration file and real-time system, we can put them together. First, deploy the configuration file generated in step 11 of the editor development process to the location specified in step 12 of the runtime development process.
2. Load *Single Engine Main.vi* in your runtime project, and press run.
3. On the front panel, you should see your input value changing and you should be able to control your output value, using the CVT monitoring loop.  
   **

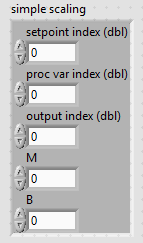
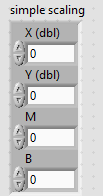
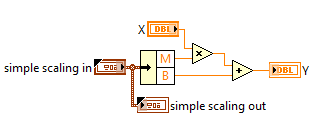
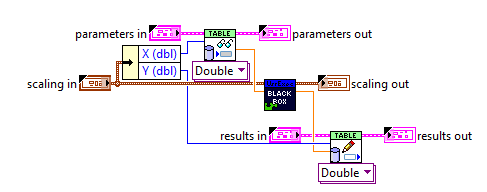
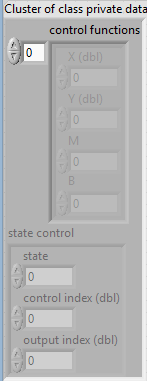
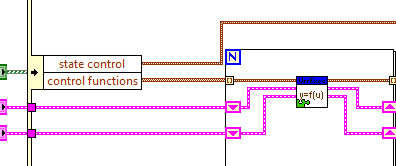
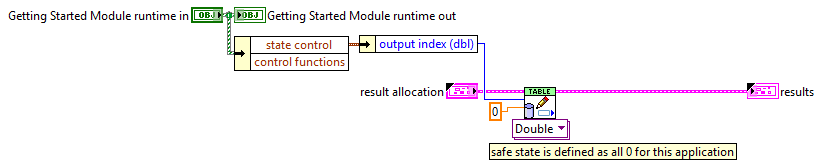
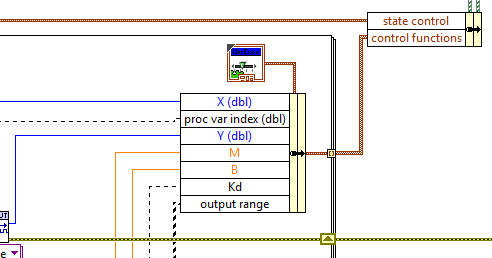
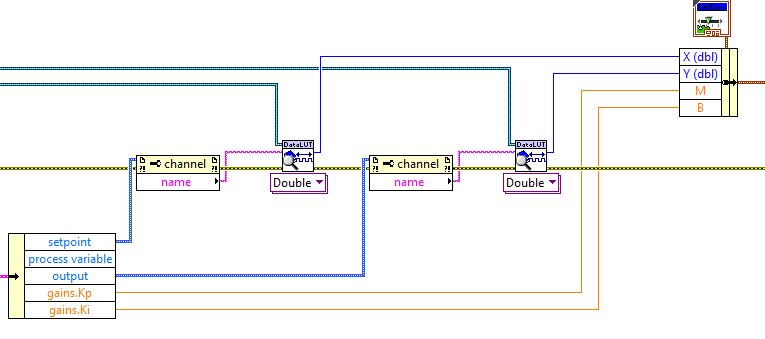
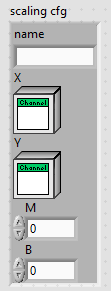
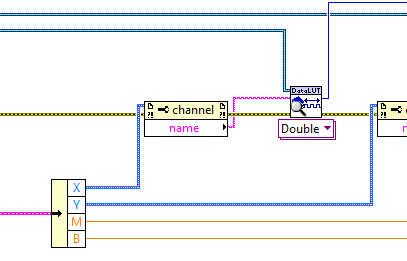
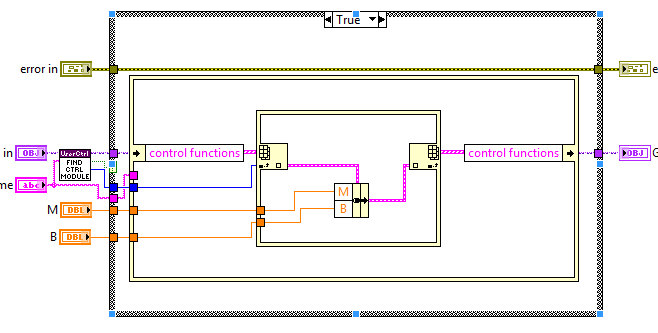
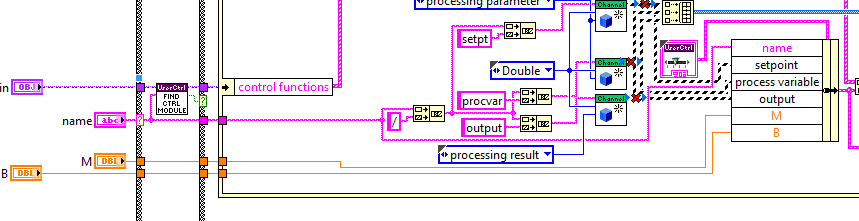
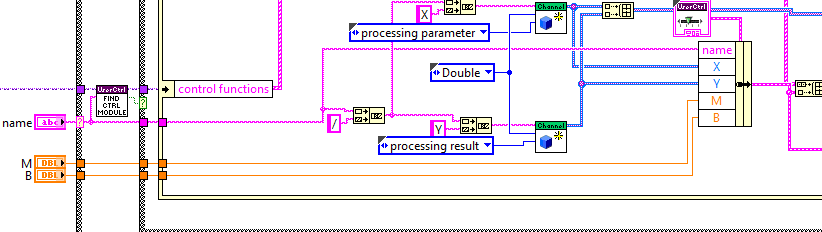
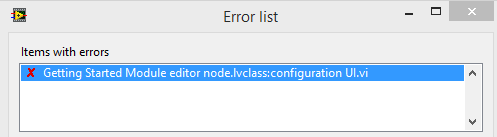
# Developing a New User Code Module

1. From the LabVIEW getting started screen, press the **Create Project** button.
2. From the dialog, select **Sample Projects** from the tree on the left hand side.
3. On the right, *User Tag Bus Module with Editor Node* should appear near the top of the list of available sample projects. Select it and press **Next**.
4. Fill in the fields for project name, and location. Select the number of instances allowed. This number will usually be either -1 (no limit) or 1 (there should only be a single instance of the class.   
     
   Select the supported targets. For most user code, which only contains logic and no I/O, all targets should be selected. For this guide, add all the targets.  
     
   For supported types, select ones which make sense for your application. For example, if you are using Modbus as a communication protocol than supporting U16s is a requirement. If possible, it is recommended that all code support at least all scalar values except strings. For this guide, just select double and I32.   
     
   Once you’ve configured your initial settings (these can be changed later), press **Finish**.  
   

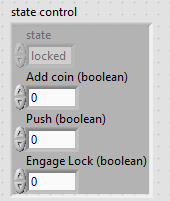
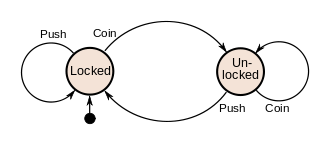
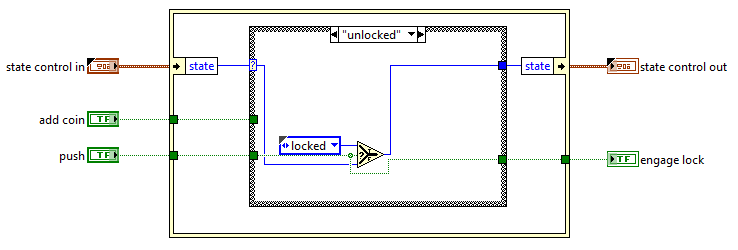
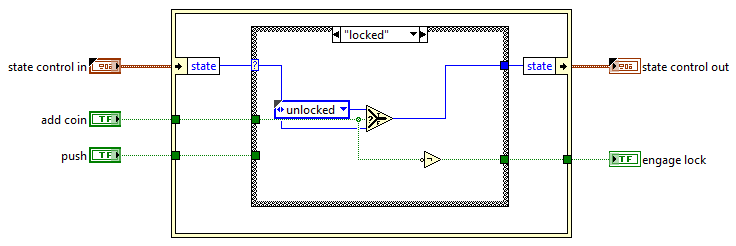
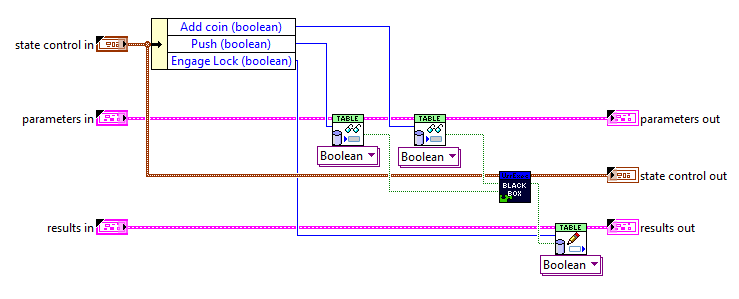
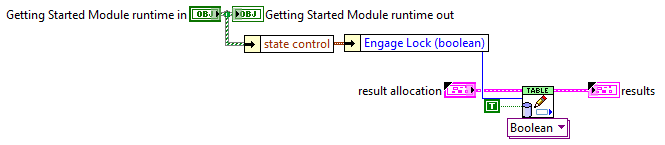
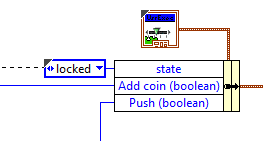
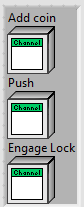
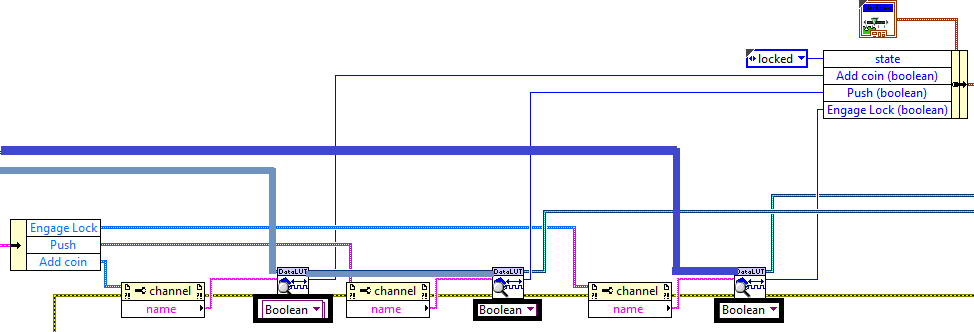
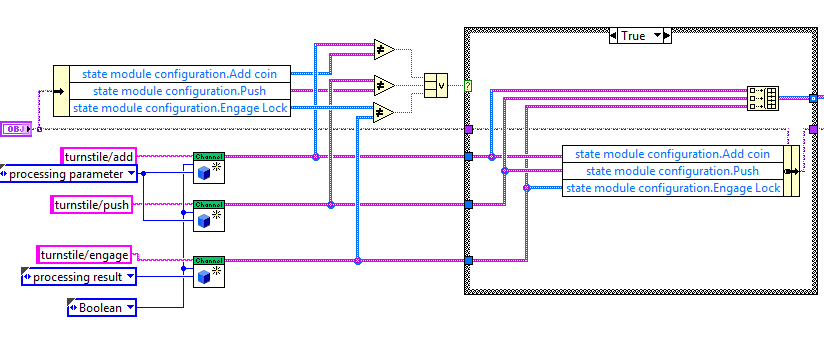
### Introduction to Development

1. In general, development of new modules should follow this pattern:  
     
   Developing the runtime informs how to develop the configuration, and then you test both. Finally, the editor consumes the developed configuration class. In most cases there is some back and forth, but the general process still moves forward as shown.
2. The sample project contains a number of bookmarks, comments which have hash tags encoded in the text. To view all such bookmarks, go to **View >> Bookmark Manager**. When you have seen every bookmark, you can be sure that development is complete.
3. Starting with the runtime, the most important functions are in the *core overrides* folder. These are the main logic components called by the engine. The other folders (*user code* and t*ype definitions*) contain VIs and type definitions used by the *core overrides*.
4. Starting with *Process.vi*, the runtime portion of the runtime code, let’s begin development. In the sample project, there are two kinds of data pre-built into the module: function data and state data. State data is used to drive some sort of state machine, and has a fixed set of inputs and outputs. Function data is code which is used in a bulk manner – that is, you might have 10 PID functions, and each PID function has a set of specific inputs, outputs, and parameters. For the purpose of this guide, we will use both.

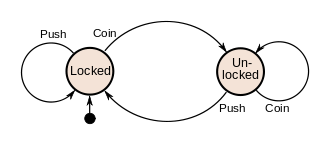
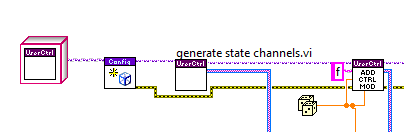
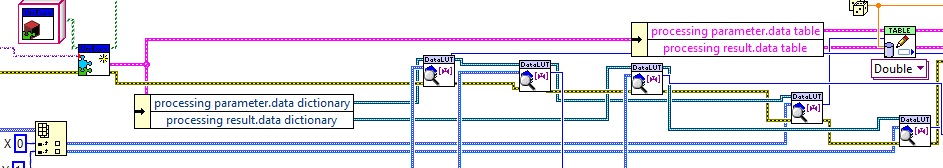
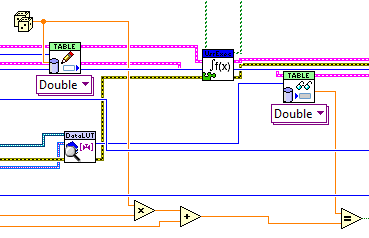
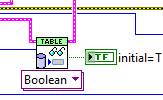
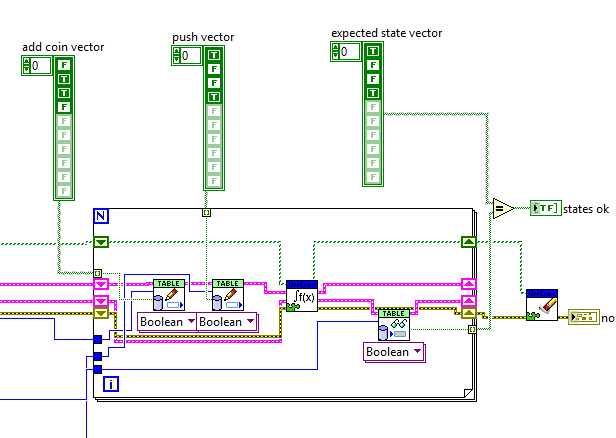
### Implementing Function-Oriented Code

1. Since LabVIEW is a dataflow language, we should start with our data. Navigate to *type definitions/control module.ctl* and rename it to *simple scaling.ctl.*
2. Open *simple scaling.ctl* and take a look at the data there. We have a few *data indices*, and then some *parameters*. *Data indices* are the variables of our function block. *Data indices* are *always* I32s, since they reference specific values in the data table. To reduce errors and simplify understanding, the *data indices* are always labeled with their data type. *Parameters* are constants, set during configuration. *Parameters* can be anything you need for your module to get the job done.
3. Since we are doing scaling, we need Y, M, X, and B. M and B are *parameters*, while Y and X will be represented by *data indices*.
4. Delete all current *parameters* (output range through last error) and drop two new numeric controls into the type definition cluster. Name these controls M and B.  
   
5. Next, delete “output index (dbl)” and rename the first two elements of the cluster to “X (dbl)” and “Y (dbl)”.   
   
6. Next, we’ll need to modify some code. Navigate to the *user code* folder and rename *simple control module.vi* to *simple scaling module.vi* and rename *black box control.vi* to *scaling function.vi.*
7. Open *scaling function.vi*. You’ll see here that we have a set of inputs and outputs which map to the *data indices* we just replaced. That is, previously we had a *data index* “setpoint index (dbl)” and on the connector pane of this VI, we have a double called “setpoint”. Following this pattern, rename “process variable” to “X”, “output high” to “Y”, and delete both “dt” and “setpoint”. Now we have a terminal “X” which maps to *data index* “X (dbl)”.
8. Go to the block diagram and remove all of the dead code. In this function we simply want to unbundle “M” and “B” from our cluster and scale “X” into “Y” with the formula Y=MX+B. The result is shown here:  
   
9. Now, open *simple scaling module.vi*. Here, we simply read data from our *data table* wires, write results back to the output wire, and call our *scaling function.vi* (image is “black box”) to process the data. Since we have removed many of our inputs and outputs, the code should be a lot simpler:  
   
10. Open *process.vi*. You can see here that the original module had a time value which is no longer necessary. Feel free to delete this code, as well as removing the “t(n-1)” field from the class.  
    
11. Next, we ensure that *process safe state.vi* is still functional. As the name implies, the engine calls this function when it enters a safe state or when the system shuts down. This gives your code the opportunity to set output values. To make this function match our new code, we should simply delete the “for” loop which sets the output value of our processing function. After all, it is unlikely that the scaling function is being fed directly to an output, so it doesn’t matter what value we set it to.  
    
12. Now that the runtime code is complete, we should adjust *init.vi* to handle the new data types. First, we should adjust our bundle functions to remove unused data and to show the data types we care about.  
      
    You’ll notice we adjusted the old *data index* input names (“setpoint” and “proc var”) to the new input name “X”, while we adjusted the old output *data index* to “Y”, our new output. This is because the indices are pulled from separate data dictionaries, so the code is already in place if we leave the locations as shown above.
13. Go ahead and delete the unused *look up tag by type.vi* function, as well as any unused wires.  
    
14. Now, we gave a problem. We have values from the “unbundle” on the left that don’t match our new simple scaling function’s needs. This is because the *init* method is the bridge between configuration and runtime data, and the data on the left is our configuration data. To fix this, go to the configuration class an open *control cfg/control module configuration.ctl*, after renaming it to *simple scaling configuration.ctl* and deleting the associated *gains.ctl*, which is no longer used.
15. You should notice many similarities between *simple scaling.ctl* and *simple scaling configuration.ctl*. Just as you deleted or renamed many items in the configuration, we’ll do the same here. Remove the constants and replace them with M and B, and change the channel object names to be “X” and “Y”, deleting the unused “process variable” object.  
    **
16. Returning to *init.vi*, we can see that the “unbundle” function has broken. Lets replace the old data with the new, and we should get this:  
    **  
    Note: Be absolutely sure to match up the channels with their appropriate dictionary. Since we know “X” is always a *parameter*, we use the *processing parameter data dictionary*. Since “Y” is always a *result*, we use the *processing result data dictionary*.
17. Our *init.vi* function is still broken because the configuration is broken. To correct this, we’ll need to fix these VIs:
    1. Add control module.vi
    2. Modify control module.vi
    3. Control function to channels.vi
18. *Control function to channels.vi* is a straightforward method. It is intended to collect all channels related to a control function – in this case, X and Y. Removing the unused channels and adding our new channels results in this:  
    **
19. *Modify control module.vi* is an API call which allows the user to change *parameters* (M and B). Simply replace the connector pane terminals (“gains” and “output range”) with single numerics (“M” and “B”), and adjust the “bundle” function to point to the appropriate items. Be sure to set the terminals as required if they are.  
    **
20. Finally, *add control module.vi* is the most complex change. First, replace “gains” and “output range” as in the previous step.  
    **
21. Next, we will need to change the properties of the channels we generate for each function. In our case, we decided the *data indices* would be doubles, so the data type may remain the same. In addition, we know the channel named “X” is a *parameter* while “Y” is a *result*. Making these tweaks to the code results in this:  
    **  
    Note: The channel names are generated relative to the function name, which is expected to be unique. This is the purpose of the other case structures, to ensure the function is named and that no other function exists with this name.
22. **Right-click** on your configuration class in the project and select **Show Error Window**. At this point, you should see only one error – with the editor node.   
    **  
    This is expected and acceptable for now. Returning to *init.vi* in the runtime class, you’ll note that the class is no longer broken and can run fine. This is a significant advantage which comes from separating the runtime and edit-time classes using a separate configuration class. Issues with one don’t affect the other.

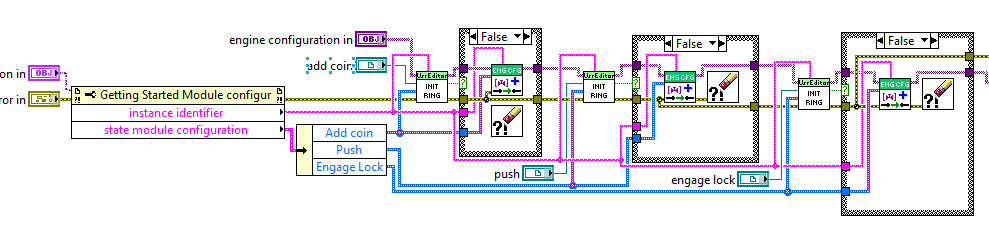
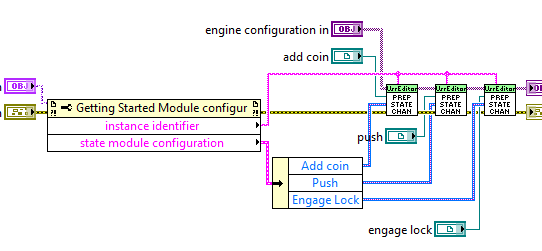
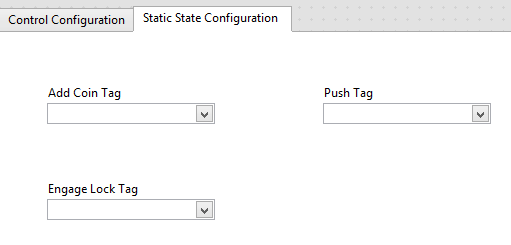
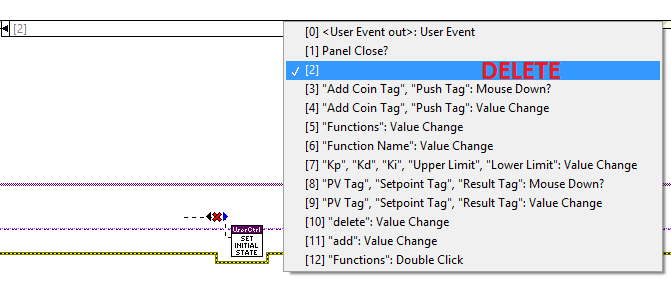
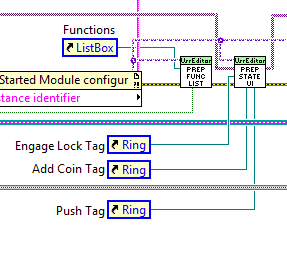
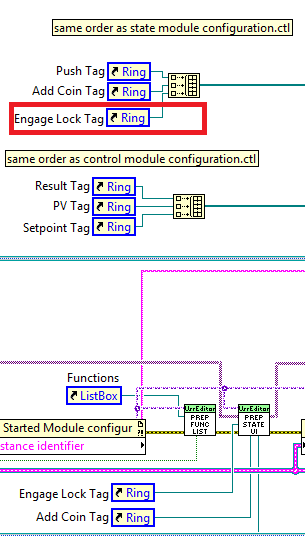
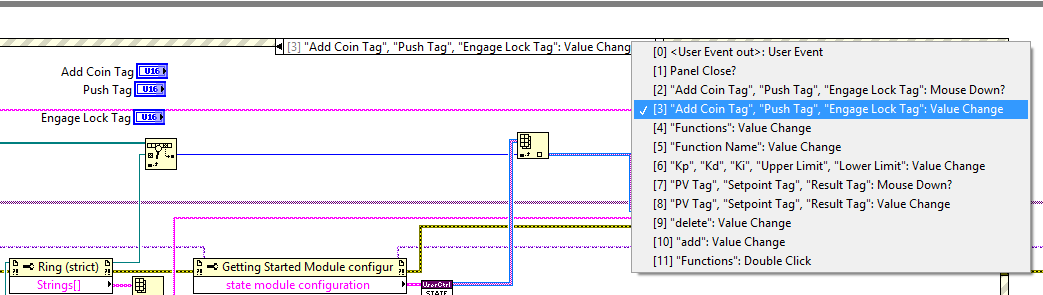
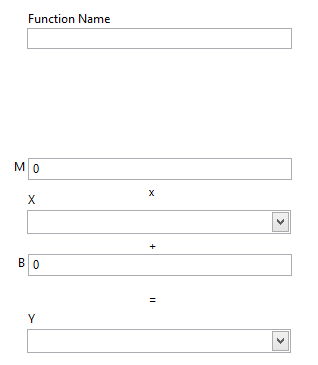
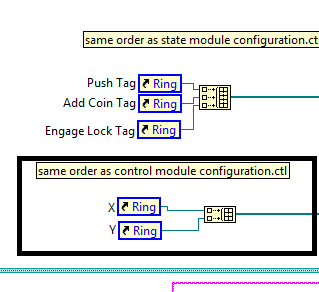
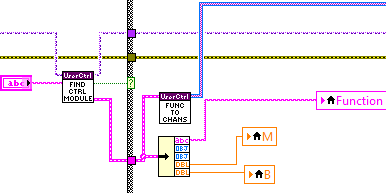
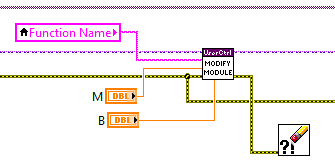
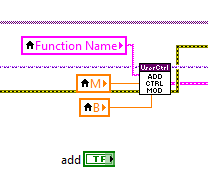
### Implementing State-Oriented Code

1. Returning to the runtime, we can begin working on the state-oriented code. Open *type definitions/state control.ctl*. This time we are going to make a simple state machine example. First, replace the “state” value with an enum and convert it into a type definition. Add the following states to the enum:
   1. Locked
   2. Unlocked
2. Add this new enum to the runtime class – we don’t need to know about it at edit-time.
3. Remove “control index (dbl)” and “output index (dbl)” and add three new numerics (representation I32) to *state control.ctl*:
   1. “Add coin (boolean)”
   2. “Push (boolean)”
   3. “Engage Lock (boolean)”
4. After adding these values, the final result should look like this:  
   **
5. Now, rename *user code/black box state code.vi* to *turnstile state machine.vi* and open it.
6. As with the function-oriented code, we’re going to add controls to this VI for every *data index* in our cluster. In this case, we need two boolean inputs (“add coin” and “push”) and a single boolean output (“engage lock”). Add these to the connector pane.
7. Now, we simply have to build the guts of our state machine. Generate a simple state machine following this state transition diagram, recalling that in this framework all code must be non-blocking:  
   
8. The final result should look something like this:  
     
   Remember, for us to detect *events* like adding a coin, we really should be detecting value changes – for example, a change in the value of “coin count” or the “add coin” boolean transitioning from False to True. However, this turnstile example only has two behaviors and so we can ignore the concept of *value-change events*, for now.
9. Rename *state control module.vi* to *turnstile control module.vi* and open it.
10. As before, we’ll need to read the *parameters* of our control function, pass them to the control function, and then write the *results* of that function out to the rest of the system. We do this using the *data indices*:  
    
11. Now that we’ve completed the main control functionality, open *process.vi* and confirm that there are no errors. Then, open *process safe state.vi*.
12. In *process safe state.vi*, we need to set the safe state for the system in case of an error. Let us assume that our safe state is “locked”, so we want to make sure that the data associated with *data index* “Engage Lock” is set to True:  
    
13. Finally, we need to make sure the code initializes to the correct state. The template sets this as a configuration option, but we know the initial state should always be “locked”. Open *init.vi* and modify the “bundle” which accepts the initial state information to have a fixed initial state: “locked”.  
    
14. At this point, we should have no errors in either the runtime or configuration classes, but remember that no errors does not mean correct functionality. We need to make sure init.vi works correctly, but to do so we first have to update our configuration class.
15. First, we should update our configuration cluster that is paired with our runtime control, *state control.ctl*. Open *(fixed) state config/state module configuration.ctl*. Here, we need to make sure our configuration for *data indices* and *parameters* matches our runtime configuration.
16. Delete “initial state”. We set that as a constant for this application.
17. Add an additional channel and rename the three channels as follows:
    1. “Add coin”
    2. “Push”
    3. “Engage Lock”  
       
18. Returning to *init.vi*, adjust the state module initialization section to locate all necessary *data indices* (“add coin”, “push”, and “engage lock”), by the correct type (boolean, not double), from the correct dictionary (parameter, parameter, and result). The code should look like this, in the end:  
    
19. Now we should update the configuration class VIs to match the new type definition. First, delete *set initial state.vi*. It isn’t necessary, now that initial state is fixed.
20. Next, open *state cluster to channels.vi* and, just as we did with the function channels, adjust the VI to return *all* channels associated with your state machine configuration.  
    
21. Next, open *generate state channels.vi*. This function is responsible for ensuring that the code module has all the channels configured correctly. First drop down a new instance of *create new channel*, and then follow the existing pattern to check if the configuration for “Engage Lock” is correct and, if not, set the value for “Engage Lock”. In addition, adjust the names and types of the other channels.  
    
22. At this point, we’ve updated everything on the runtime and configuration classes, so we should test. The editor is polish, and we don’t want to polish the code before we’ve confirmed that we have the functionality we want.

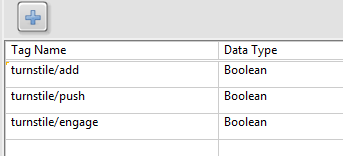
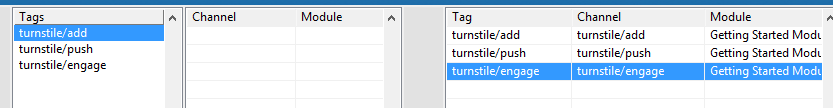
### Testing the Module

1. The first thing we should do is get the core configuration tests up and running – serializing, adding and removing channels, etc. To do this, we need to have a basic configuration to test with. Open *<Your Module Name> tests.lvlib:create basic config.vi.*
2. First, we need to update the function generating loop to match our new type definitions. It is already creating a unique name for each function, but we need to produce a valid value for M and B. For the purposes of testing most functions, random numbers will suffice.
3. Once you’ve generated reasonable values for M and B, the function should no longer be broken. Run the following tests and confirm that they pass:
   1. *Channel operation tests.vi*
   2. *Don’t recreate channels test.vi*
   3. *serialization tests.vi*
   4. *init runtime test.vi*
4. *Serialization tests.vi* is unlikely to pass, because we are flattening a random double precision value to an ascii string and reading it back. Replacing the random numbers in step 54 with constants will result in a test which passes. For now, we can leave the tests as they are, but it would be better in the future to adjust the *to string* and *from string* functions to make sure they save values in a format which gives us the precision we require for our application.
5. Open *runtime io test.vi*. This function is still broken. This is fine, as we want to confirm the functionality of our system. Use the API functions of your configuration class to create a test which tests the state-oriented and function-oriented requirements of your code:
   1. Y=MX+B should hold true for every configured function
   2. This state diagram should hold true when using the front panel to adjust values for “add coin” and “push”:  
      
6. Our test has the following features:
   1. Create a set of state machine channels and function channels:  
      
   2. Initialize, and locate these channels in the data sets:  
      
   3. Run *process.vi* and confirm that Y=MX+B for the configured function:  
      
   4. Confirm the initial state:  
      
   5. Provide a set of test vectors for the state machine and run *process.vi* for every vector:  
      

### Developing a Configuration User-Interface

1. Now that we have confirmed that the runtime components work properly, we can work on the UI. Remember, the module can now be configured completely on the block diagram. The idea is to allow people to use the built-in configuration editor to configure their system.
2. Open *configuration UI.vi* in the editor class and explore the code. First, we initialize everything (which gives an opportunity to generate “fake” modules and engines for debugging purposes). Then, we prepare the UI with the values in our system (display channels, tags, functions, or any other values we need to configure). Finally, we enter a simple event loop. This event loop is entirely under your control, except that you should be sure to handle the “stop”, “save”, and “refresh” user events. The template does this for you.
3. Open *ui updaters/prep state ui.vi*. We have a larger number of channels now, so we need to simply expand the VI to support our third channel. There are more complex ways of updating the UI, but the code as-is simply looks at each channel, figured out which tags can be mapped to them, and immediately forces generation and mapping of a new tag. That is, if we have a channel “Add coin”, this function finds all boolean tags in the system, determines if the channel is already mapped and, if not, makes a new tag “Add coin”. This makes sense for the state machine use case, as we *always* want to have these values in the system.
4. Rename “input tag” to “add coin” and “output tag” to “push”, and then create a new reference of the same type (ring control) and name it “engage lock”. Delete “initial state” and the associated property node, as we removed this from our configuration. Finally, make a third copy of the ring and mapping code and use it to initialize the “engage lock” reference. While messy (block diagram clean-up), the code is fairly straight forward. Since it is duplicated, you can make each module into a subVI if desired.  
     
   
5. Now, return to *configuration UI.vi*. We need to perform the same task (rename input and output, add a new item for “engage lock”) on the second tab, *Static State Configuration*. Remember to delete the “Initial state” control and the associated event case, as well.  
     
   
6. You’ll also need to find the instances of *prep state ui.vi* and provide them with a reference to the new control, as shown:  
   
7. Along the same lines, concatenate a reference to the new “Engage Lock Tag” to the array of references with the note “same order as state module configuration.ctl”.  
   
8. Finally, modify the mouse down and value change events associated with “add coin tag” and “push tag” to include the same events for the new “engage lock tag” control.  
   
9. Now, on the front panel, move to the *Control Configuration* tab. The easiest way to update this code is to change the names of “Kp” and “Kd” to “M” and “B”, and then delete “Ki”, “Upper Limit”, and “Lower Limit”. Then, rename “Setpoint Tag” to “X tag” and “Result Tag” to “Y tag”, deleting “PV Tag”.  
   
10. Next, we need to update the references: rearrange the X and Y references to match the order of *control module configuration.ctl*, now *called simple scaling configuration.ctl*. This order is X, Y.  
      
    Just as with the other tags, this makes sure we can associate a channel *object* with a front panel reference. Keeping them in the correct order is the simplest way to do so.
11. Go to the *Function Name*: *Value Change* event. Remove the unused local variables and make sure to bind the function values “M” and “B” to the appropriate locals.  
    
12. Go to the event *“M”, “B”: Value Change*, and tie M and B to the Modify Module function.  
    
13. Go to the event *“add”: Value Change* and again, attach the M and B values to the Add Control Module function.  
    
14. Go ahead and run your code. It should function in the development environment and allow you to debug any issues.

### Generating a Configuration in the Editor

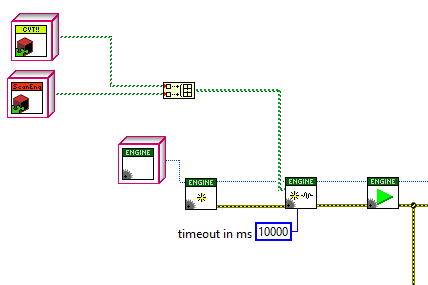
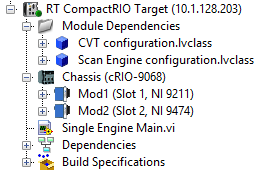
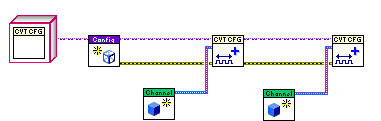
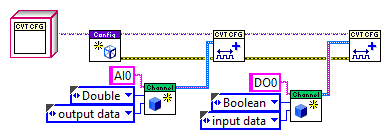
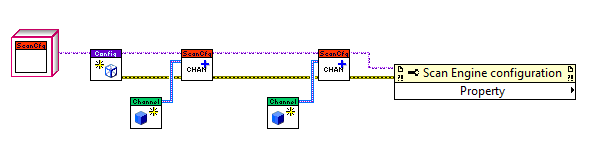
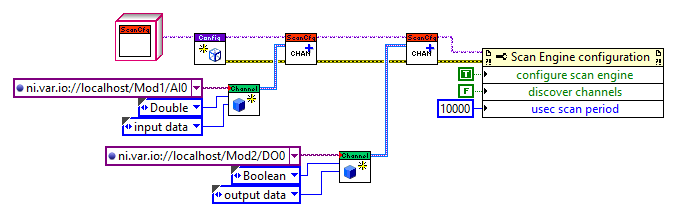
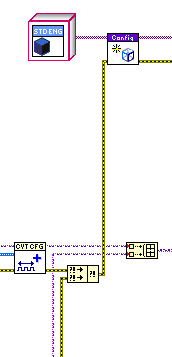
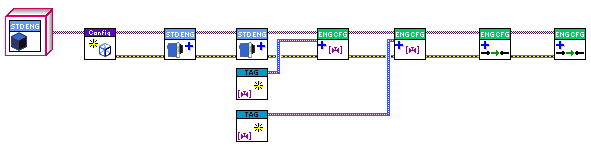
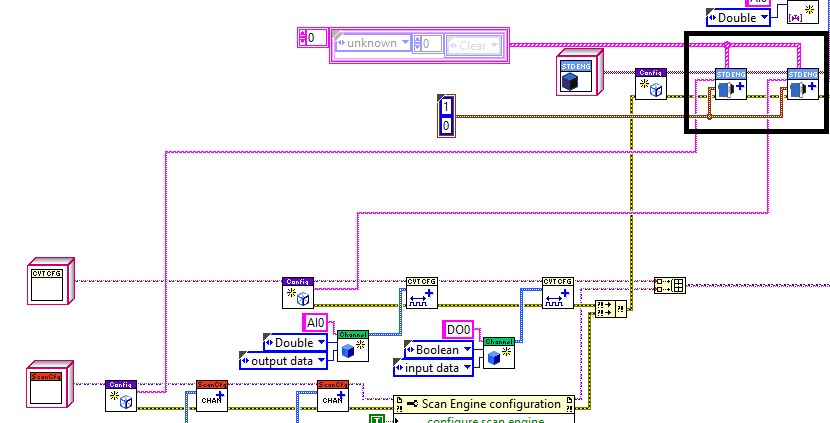
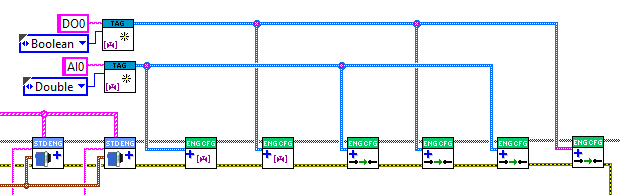
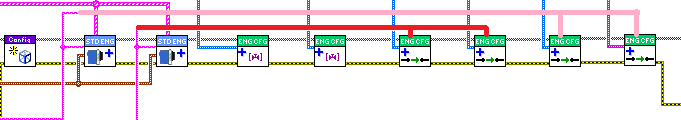
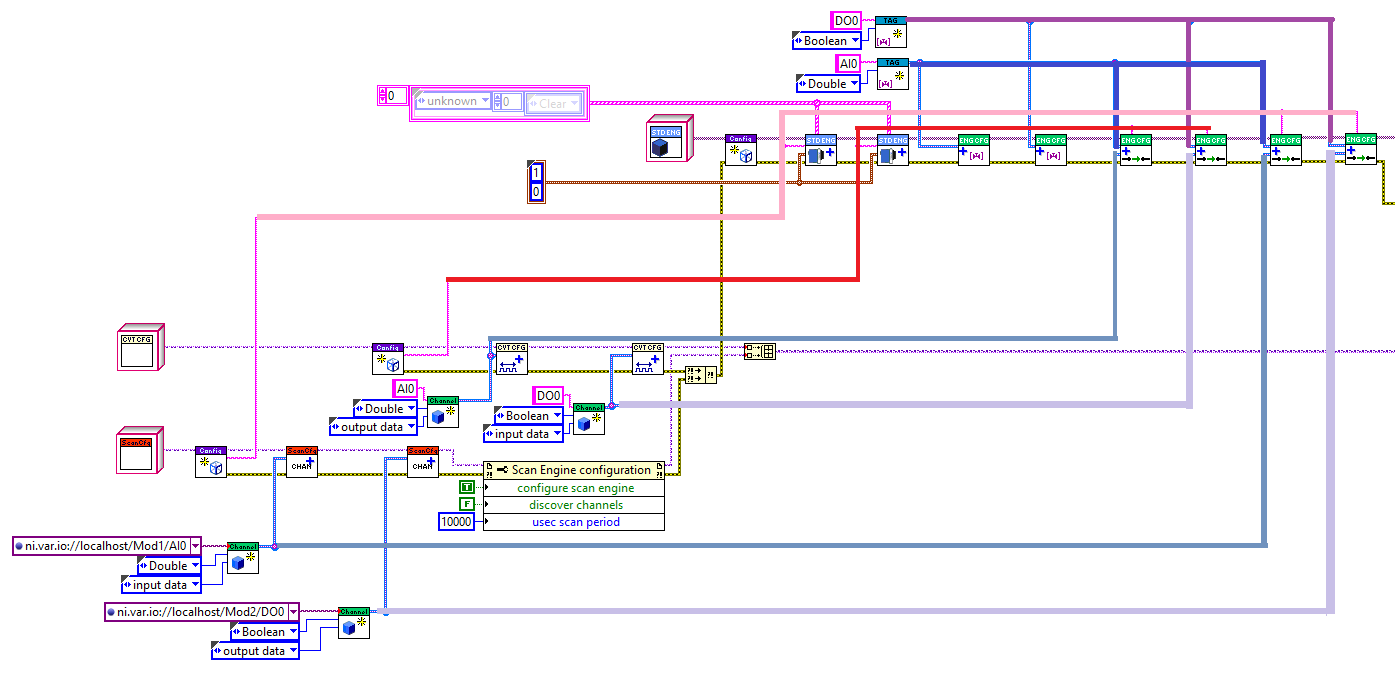
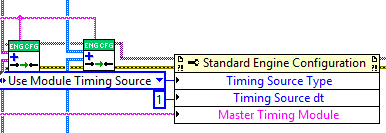
1. Load your editor and press run.
2. In **Tools >> Edit Search Paths**, press **Add** and navigate to the folder containing your new editor module. Then, press **OK**.
3. Add a cRIO, Standard Engine, and your module.
4. Navigate to the *Tags* view and confirm that your state machine tags were generated automatically:  
   
5. Navigate to the *mappings* and confirm that all mappings were generated automatically:  
   
6. Test out the rest of your module, using the *Tags* and *Mappings* views to confirm functionality.

# Developing a New I/O Module

# Advanced Topics

## Creating a Hard-Coded RT Application

The TBDF can be used without a configuration editor by simply configuring the system on the RT VI diagram. This section covers development of a hello world application without the configuration editor.

1. Follow all steps above for the Single Engine RT Project.
2. Add instances of the configuration classes associated with each runtime class:
   1. Navigate to *vi.lib\NI\TBM Modules\CVT Access\module\configuration* and drag CVT configuration.lvclass onto the block diagram.
   2. Navigate to *vi.lib\NI\TBM Modules\variable\module\configuration* and drag Scan Engine configuration.lvclass onto the block diagram.
   3. Navigate to *vi.lib\NI\* *TBM Engines\Standard Engine\Configuration* and drag Standard Engine Configuration.lvclass onto the block diagram.
3. Delete the first three VIs (with the PLCfg header) and associated constants and nodes:
   1. Read full file.vi
   2. String to system.vi
   3. Unbundle “targets”
   4. Index “targets”
   5. Unbundle “engines”
   6. Index “engines”
   7. Load runtime from configuration.vi
   8. Unbundle “engine configuration” and “modules”
4. Build an array of runtime modules – CVT and Scan Engine in our case – and wire them to the “modules” input of Initialize.vi.
5. Wire the standard engine runtime constant to “Engine Runtime Interface in” terminal on the engine’s Open.vi. At this point, the code should look like this:  
   
6. To configure the system, we will have to create tags, create channels, and then create mappings.
7. For each configuration object, wire it to an instance of **Addons >> NI Tag Bus Module Framework >> Root Configuration >> Initialize Configuration.vi**.
8. Because we are configuring the project manually, we will need to use the configuration objects as our API. For ease of use, drag the module configuration classes from dependencies into your project.  
     
   The engine configuration has an API on the palettes, so this will not be necessary for that class.
9. For the CVT configuration object:
   1. Drop down two instances of *add channel to CVT configuration.vi*
   2. Drop down two instances of **Addons >> NI Tag Bus Module Framework >> Channel >> Create New Channel.vi**.
   3. Wire these up as appropriate:  
      
   4. For the first channel, select names and data types which match the modules you have available. As before, we will use the following settings:
      * Name: “AI0”
      * Type: “Double”
      * Classification: “Output Data” (Note: we are copying data from our AI0 tag into the CVT, meaning it is an output TO the CVT module).
   5. For the second channel, select names and data types as appropriate. We will use the following:
      * Name: “DO0”
      * Type: “Boolean”
      * Classification: “Input Data”
   6. The final configuration should look like this:  
      
10. For the Scan Engine configuration object:
    1. Drop down two instances of *add channel to scan engine configuration.vi*, a property node, and two instances of *create new channel.vi*.
    2. Wire these up as appropriate:  
       
    3. For the first channel, select settings which make sense for your input channel. The scan engine module requires a full variable URL as the name. For our configuration, we selected the following:
       * Name: “ni.var.io://localhost/Mod1/AI0”
       * Type: “Double”
       * Classification: “Input Data” (Note: we are copying data FROM our scan engine AI0 value into the tag AI0, meaning it is an input FROM the scan engine module).
    4. For the second channel, select settings which make sense for your output channel. For our configuration, we selected the following:
       * Name: “ni.var.io://localhost/Mod1/AI0”
       * Type: “Boolean”
       * Classification: “Output Data”
    5. On the property node, select “configure scan engine”, “discover channels”, and “usec scan period”. Change all of these to write and set the values to True, False, and an appropriate scan period. We selected 10000 usec.
    6. The final result should look like this:  
       
11. Use a build array function on the configured scan engine and CVT objects in the same order as step 4, and pass that into the *module configurations* input of *initialize.vi*.
12. Merge the error wires from configuring the two modules and pass the result into the instance of *initialize configuration.vi* attached to the standard engine configuration.  
    
13. For the standard engine configuration:
    1. Drop down two instances of **Addons >> Standard Engine Module Execution System >> add module to engine.vi**
    2. Drop down two instances of **Data Communication >> NI Tag Bus >> construct tag.vi**
    3. Drop down two instances of **Addons >> Tag Bus Execution Engine Interface >> Tag API >> Add Tag.vi**
    4. Drop down four instances of **Addons >> Tag Bus Execution Engine Interface >> Mapping API >> add mapping.vi**
    5. Drop down a property node
    6. Wire these together and to the engine configuration in this order. The property node is not shown, but should be placed at the end.  
       
    7. Create tags with the appropriate types. In our case, the names do not actually matter. For simplicity we use the following settings:
       * Tag Name 1: “AI0”
       * Tag Type 1: “Double”
       * Tag Name 2: “DO0”
       * Tag Type 2: “Boolean”
    8. Wire the instance identifier for each module (provided by *Initialize Configuration.vi*) to the instances of *add module to engine.vi*. Use default values for the other inputs.  
       
    9. Wire each tag to two instances of *add mapping.vi*, alternating between instances.  
       
    10. Pass an instance ID to the *Owner ID* input on each pair of *add mapping.vi* functions. The pairs should be sequential. The idea here is that we need to generate a mapping between each module and each tag.  
        
    11. Wire the CVT channels into the *add mapping.vi* functions which have the CVT instance ID string passed in as the *Owner ID*. Do the same for the Scan Engine channels. Be sure to tie the “AI0” channels to the “AI0” tags, and the “DO0” channels to the “DO0” tags.  
        
    12. On the property node, select “Timing Source Type”, “Timing source dt”, and “Master Timing Module” and change all to write. Set the timing source as “Use Module Timing Source” and the dt as “1”. Wire the instance ID of the scan engine module to “Master timing source”.  
        
14. Wire the configured instance of the standard engine configuration object to the *Engine Configuration* input of *Initialize.vi*.
15. Pass the error wire from the last *add mapping.vi* instance to the *error in* input of *Open.vi*.
16. Press run. The system should function without errors.