# Creating New DCAF Modules

Before developing a new module, it is best to have a basic design for each module in the application. Modules should be designed to have as much reuse value as possible while balancing performance needs and the cost of implementation. While it is possible to include input, processing, and output functionality within a single module, splitting the I/O and processing into separate modules may result in two more flexible modules that can more easily be reused. Alternatively, it is likely significantly faster (at least in the short-term) to build one fairly rigid module that does everything. Just like other types of design work, module design in DCAF can be viewed as an art where there isn't always a clear right answer.

When designing a module, there are a few important questions to answer.

* Will the module implement the Input, Process, or Output methods?
  + I/O modules like the Scan Engine Module will need Input and Output methods.
  + Processing Modules for control algorithms may just need a Process method.
* For each method, are the needed channels static or dynamic?
  + The Scan Engine Module is dynamic because the user specifies how many channels to use and how to configure each.
  + A control algorithm can often be implemented as a static module because the numbers of inputs and outputs are known when it is being built. If a new module can be made using a static list of channels, decide the name and data type for each.
* What information needs to be configurable?
  + The Scan Engine module needs to be able to configure the channels needed as well as the Scan Period
  + A control algorithm might configure the control gains required for a particular application.

After the basic design for each module is complete, decide which of the two module templates to use for each. This decision is usually as simple as whether or not the module uses Static or Dynamic channels for each of its methods.

Typically a module for application-specific processing logic would have a static number of inputs and outputs for its Process method. In that case, it would be recommended to start with the Static Module. While in the case of an I/O module that will be reused in multiple applications, the number of I/O channels per application depends on the DCAF configuration and is therefore Dynamic. In this case the Dynamic Module template is likely the best choice. The Dynamic Template is much more flexible, but this comes at the cost of significantly greater complexity. The Static Template is designed to be as easy to use as possible.

This guide will first cover the process of creating a Static Module, and then go over a Dynamic module. Note that both are templates for creating a DCAF Module, and that there is therefore significant overlap between the two. They both rely on the same underlying set of code, and it is possible (though not recommended) to build a new module from scratch by inheriting from DCAF’s base module class. For both templates, the majority of the development work will happen in a set of virtual folders called “Customize”. When in doubt as to how to add something to the module, check these folders first.

When the scripting tools are used to create a module, they create 3 main classes – configuration, editor and runtime. This guide will call these scripted classes as *ProjectDemo configuration.lvclass, ProjectDemo editor node.lvclass,* and *ProjectDemo runtime.lvclass*. This is to differentiate it with the parent classes such as the *control module configuration.lvclass*

Worked examples can be found as part of the [hands-on session found here](https://github.com/LabVIEW-DCAF/Documentation/tree/master/Trunk/Getting%20Started%20Material/Hands%20On/Current).

The majority of changes to any DCAF module from the template should need to be done in the Customize or Override folder of the 3 classes – the other place that would likely need to change is in the class controls themselves to allow for carrying data through the runtime, for example.

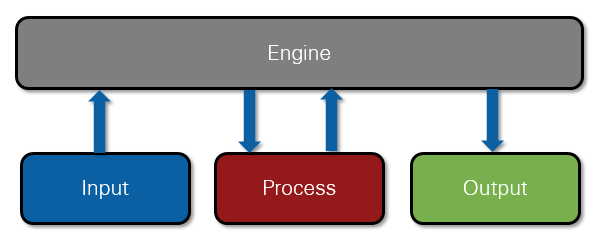
As a final note, many modules could be of value to the rest of the DCAF developer community. If willing to share the code, consider posting in the community before getting started on development and others may help. Sharing code with the community has the additional benefits of spreading out the testing and maintenance burden of the module. For more information on how to collaborate, see that chapter of this guide.

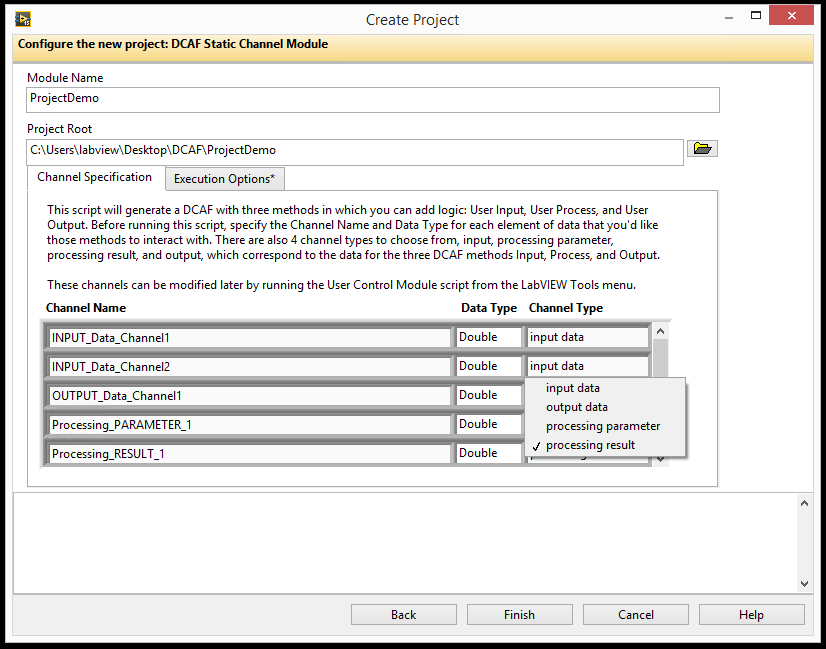
## Static Modules

### Static Module Checklist

* Create the Module from the Static Module Sample Project
  + Define the Channels
* Add Parameters to *set desired keys.vi* (optional)
* Handle initialization (optional)
  + Handle initialization of references, sessions, parameters
  + Pass References into Runtime class if necessary, modify object to allow this
* Implement the Input, Processing, and/or Output Methods as needed
  + Modify *user process.vi* to contain references or the Runtime class if needed and *process.vi* to pass data necessary to *user process.vi*
  + Add Processing code to *user process.vi*
  + Add code to *input.vi* to pass necessary sessions or config data to *user input.vi*
  + Add input code to *user input.vi*.
  + Repeat for output
* Programming Considerations for Run-State Methods
  + Determinism
  + Non-blocking execution
* Specifying other properties
* Testing and validation

### Creating the Module from the Sample Project

Before using this template, make sure you have a basic design for all modules and have a static list of channels defined for this module's Inputs, Outputs, Processing Parameters, and Processing Results. Create the module in LabVIEW from **File>>Create Project**, navigate to DCAF on the hierarchy to the left, and then select **DCAF Static Channel Module**. 



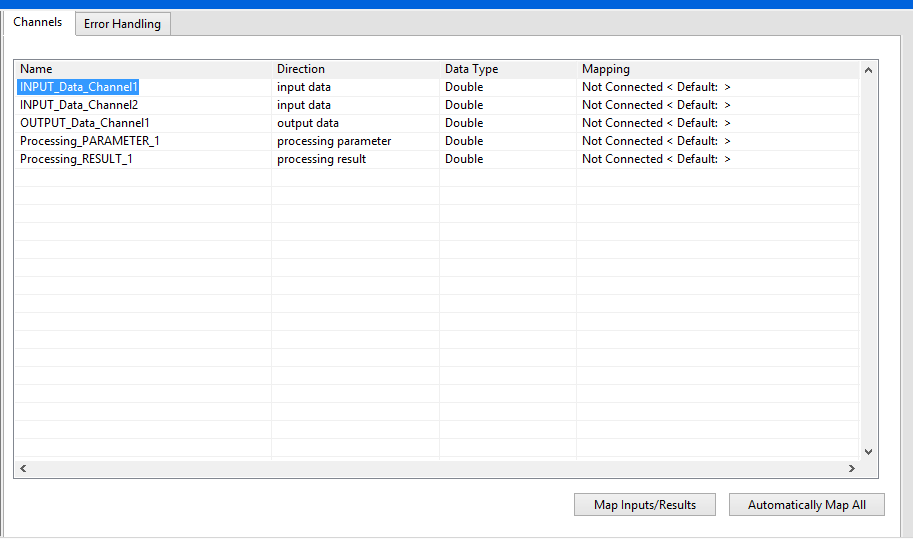
In the Create Project page, give your module a descriptive and unique name and choose a place to store it on disk. Next, add each of the static channels by name and carefully choose the correct channel type (Input data, Output Data, Processing parameter, and Processing result). It is important to pick the correct channels in this stage to avoid having to rescript the module (explained later).

On the Execution Options tab, specify the number of execution instances allowed for the module. This field can be used to prevent multiple copies of a module from being added to a configuration in cases where this may cause unwanted behavior (for example if they must share a resource like the LED module). In most cases this value should remain set to –1 for unbounded numbers of modules, in some cases 1 is appropriate, and very occasionally a different integer should be used.

In addition to the channels, static modules can have **Parameters** which can be used as configuration data. This is data that should only be read once from the configuration file at the module’s initialization. These are defined later, not in the create project dialog. Click 'Finish' to start the generation of the new module at the specified folder location.

### Modifying the UI and defining Parameters

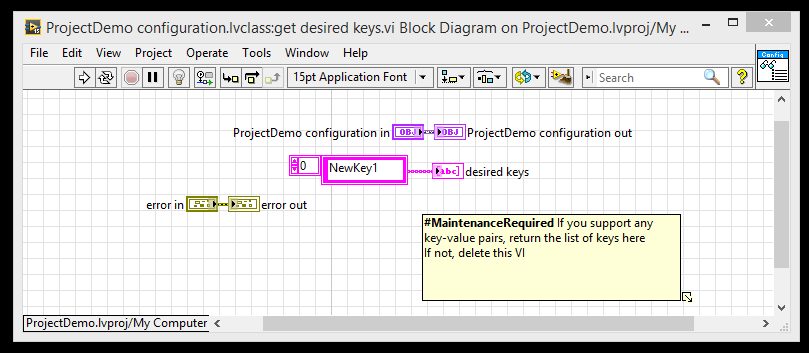
Having created the module, the UI portion should already be ready to run. Add the module to the Configuration Editor's search path and try adding it to a configuration. The built-in UI will populate a tab for Channels and one for Error handling. If no channels have been defined at creation (a rare but legitimate use-case), the tab itself will not appear. Note that there is no Parameters section in the UI by default:

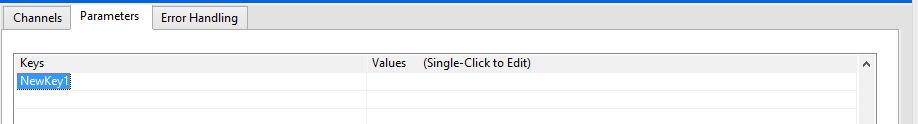


In general, **a static module's UI** **should not be directly modified**. While this is supported, it is generally better to modify the module's parameters and channels and let the provided UI update itself to reflect those changes.

Parameters, different than channels, are also an important part of the Static Channel template. Parameters allow a user to specify values on the UI used during initialization and held constant during execution. Examples include items like directory paths or timeout values. These may need to be specified through configuration, but they shouldn't be updated each iteration of the loop like a Channel.

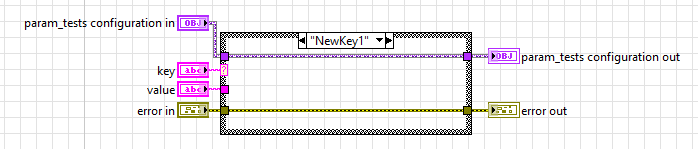
To allow the user to set Parameters, add some parameters to the g*et desired keys.vi* under the *Project Demo Configuration.lvclass*’s Customized folder. For example, note here that the “NewKey1” was added to the **desired keys** indicator, making the new Tab in the editor appear:





Having added the parameter to the *get desired keys.vi*, next modify the *validate key value pair.vi*. This VI pulls out the parameter and can be used to check whether the value for the parameter is valid. By default, any new parameter takes this VI to the default case that calls the parent method which throws an error. If there is any error in this VI the value will be erased.

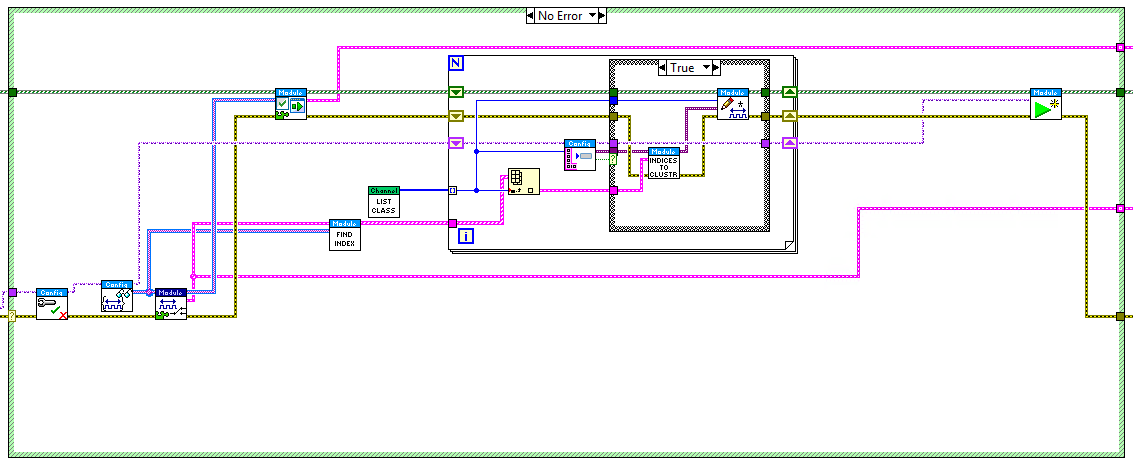
For the case of NewKey1, we could assume any string value to be correct:



Finally, create an override for the *Apply Key Value Pairs.vi* called within the *user init.vi* in order to do something with these values – please view the Modifying the Initialization section for more information. If the values are needed outside of the initialization, then modify the project’s *runtime.lvclass* to pass the values where necessary as well as the Input, Output or Processing methods such that they may use the *configuration.lvclass*.

### Modifying the Initialization

In Static modules, the Init VI is not directly modifiable. Instead, the Engine calls the Parent Class (*Control Module Runtime.lvclass) init.vi:*

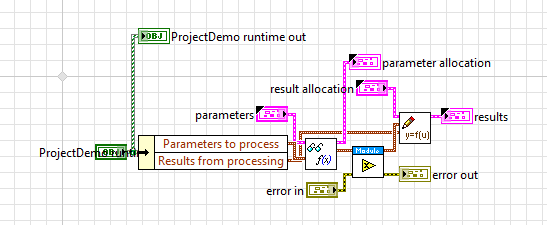


This VI takes care of getting the indices of the tags that the module needs and calls the *user initialization.vi*. To have a custom initialization scheme for something that needs to be done only once, like opening references to the hardware, override that VI and use the *Call Parent Method* to avoid having to replicate all its other behavior. It also contains the *Read Key Value Pairs.vi* as well as the *Apply Key Value Pairs.vi*, the second of which may need to overridden to allow for using the Parameters as defined earlier.

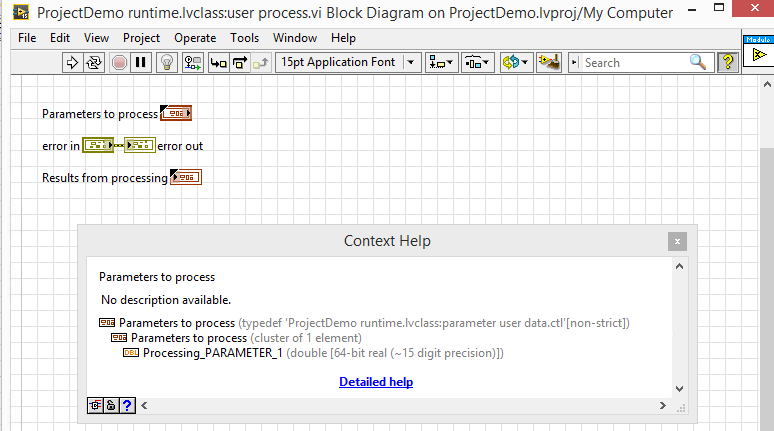
### Implementing the Input, Output, and/or Processing Methods

For each of the Input, Output, and Processing methods, the Static Channel template will generate both the main method and a 'User' version of the method that gets called as a subVI. The top-level or main method is meant to be re-scripted by the template, while the 'User' versions are meant to house custom code.

The example below shows a 'Process' method VI calling a 'User Process' subVI. The 'Process' VI includes scripted code that gathers parameters from the tagbus through a **scripted accessor** and passes them to the *user process.vi* which should contain the module’s main processing logic. Then it takes the processing output and populates the tagbus channels through another **scripted accessor**:



The *user process.vi* as created is a blank slate to put logic in. **This VI is where most of the functionality and custom programming actually needs to happen.** Note that the controls and indicators contain the parameters as defined in the creation process:



Note that changing the parameters to process and the results would require changing this TypeDef and the **scripted accessors** as well as some other code behind the scenes. If they would need to be changed, use the **Static Module Scripting Utility** found under Tools >> DCAF >> Static Module Scripting Utility when editing a VI.

Static modules, as created from the tool, do not pass the *runtime.lvclass* into the user input, user process and user output methods*.* Therefore, in order to be able to use data from the initialization case such as parameters, or other information that should not be exposed as a channel, you should modify the user input, user processing and user output methods to have the *runtime.lvclass* as an input and output. Then, from the *input.vi*, *output.vi* or *processing.vi* pass it the object. Do not modify the currently existing terminals from these VIs however.

Programming each of the Input, Output, and Processing methods follow essentially the same workflow. In most cases, some of the methods won't be needed. It is safe and recommended to delete both the top-level or main method and its 'User' subVI from the runtime class if they will never be used.

### Input, output, and process method execution performance

Whenever possible, modules should be developed to execute deterministically. This means that they can have a guarantee on the upper bound of their execution latency. Memory allocations and network communications are some of the most common causes of non-deterministic behavior. Refer to [Chapter 3 of the cRIO Developer's Guide](http://www.ni.com/pdf/products/us/criodevguidesec1.pdf) to learn more about deterministic programming practices. Also note that DCAF makes this job much easier by handling most memory allocations up front and by handling loop configurations.

If deterministic behavior for a module is not possible, the next best thing is to be non-blocking. The simplest example of blocking execution is a standard 'Wait' function. Blocking execution can be problematic because modules get called sequentially within an engine, and if one module blocks execution then everything within that engine is also blocked. Blocking behavior can often be rewritten to be non-blocking. For example, if something needs to happen after 5 seconds, use a 'Tick Count' vi to measure when 5 seconds has elapsed instead of waiting.

While not recommended, it is still possible to execute blocking code within DCAF. The Standard Engine provides an 'Asynchronous' execution feature that can prevent a blocking module from affecting other modules.

Implement the 'is module blockling.vi' Method of the module's configuration class based on how it is expected to perform.

Use the Engine's timing tags functionality to test performance. Jitter can also be measured by logging every value and measuring the worst-case variation. For best performance, configure the engine to reserve a core for execution.

### Setting supported targets and other editor options

It is possible to change the module options such as the supported types or targets after the module has been created.

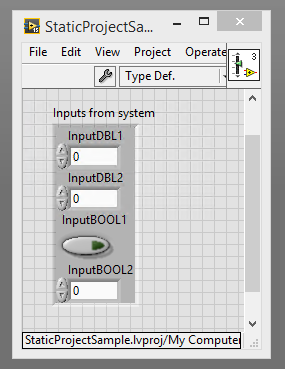
To change the data types supported by the module, change the *get supported types.vi*, a method under the configuration class. Similarly, to change the supported targets, modify the *Get Supported Targets.vi*. It is possible to get all of the DCAF supported types using the *list all targets.vi* part of the configuration class.

To load a glyph for your module in the DCAF editor, update the 'GetText.vi' in the editor node class to output the filename of the PNG picture. The picture must be in the same directory as the .lvclass file of the editor node class.

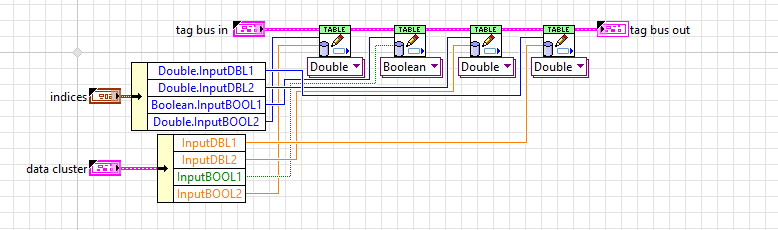
### Changing Channels on a static module and re-running the scripting tools

When a static module is created, a series of controls and accessors are scripted to avoid having the developer need to write code in order to get data out of the Tag Bus. Should the channels need to be changed, there is a tool that can be used to rescript all of the necessary controls and accessors.

First, make the change in the *ProjectSample runtime.lvclass*’ controls in the *user data* virtual folder. For example, to change the module’s inputs change the *input user data.ctl*. Note that there is an input called InputBOOL2 that was typed to a double in the module’s creation:

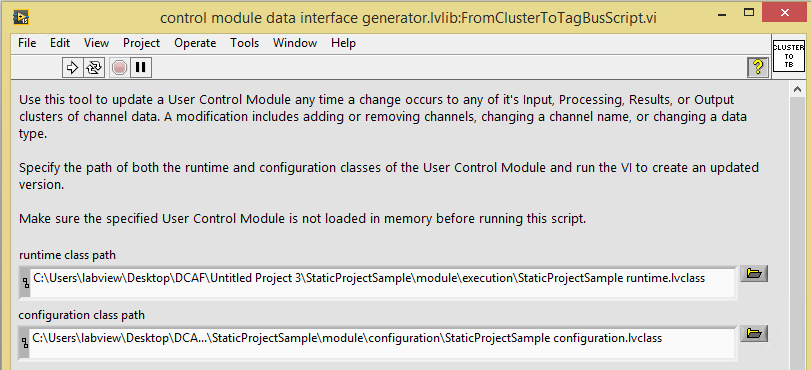


This had scripted the following accessor (*ProjectName runtime.lvclass:input to tag bus.vi*:

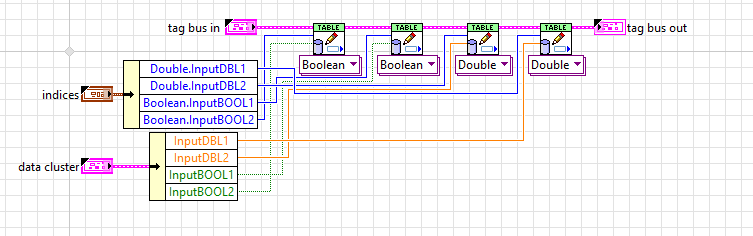


To rectify this, that control is changed to a Boolean and the scripting utility is opened (**Tools >> DCAF >> Launch Control Module Scripting Utility**). Before launching the utility, ensure that the DCAF Configuration editor is closed – otherwise the controls and accessors may be locked.

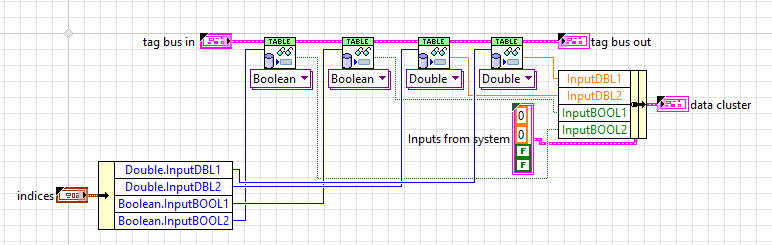
Enter the path to the Runtime class and the Configuration class and run the VI. Note – you can “drag” the class from the project onto the control which will populate the path.



The accessors were now modified to account for the different data type:



There are other options in the utility as well. You can choose to avoid scripting different channel types by toggling their **Enable** option. The Reverse option scripts the opposite accessor as well for debugging purposes. For example, toggling the **Reverse** option on the inputs will generate a vi called *input from tag bus.vi* that will use the same indices as the accessor vi:



Note that modifying or removing controls from a cluster will typically break backwards compatibility with any existing configuration files that use that Static Channel module. If still in development, this may be acceptable. However if backwards compatibility is desired, the To String and From String methods of the Configuration class must be modified manually to maintain compatibility. These methods are explained in greater detail in the Dynamic Module section.

## Dynamic Modules

The Dynamic Module template is more flexible than the Static Module template, but there is some greater cost in the form of less scripted code to take care of things. The main two would be the channel manipulation as well as the editor UI modification.

### Dynamic Module Checklist

* Create Module
* Edit UI
  + Chose parameter set
  + Modify Line control
* Create Initialization
  + Create Mappings
  + Pass Config, including mapping info to Runtime Class
* Write I/O (Optional)
  + Create I/O code
  + Write to Tag Bus through Mapping info (in runtime class)

### Creating a Dynamic Module

Through the Create Project tool, the Dynamic Module has 3 tabs: Execution Options, Supported Targets, and Supported Types.

The Execution options tab works the same as the Static module – please view the Static Module section on this.

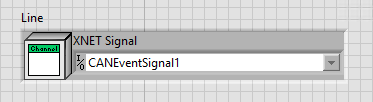
For Supported Targets, you need to tell the module whether the editor will allow a user to drop the module in an engine running on a specific type of target. This is because the dynamic modules are often used to communicate with hardware that would be target or OS dependent.

Finally, the Supported Types tab will allow options for what kind of types can be created as Channels.

### Modifying the UI and defining Parameters

Similar to a Static Module, a Dynamic Module has two main types of configuration information – Parameters and Lines. However, the Dynamic module does not know a-priori how many lines will need to be instantiated. Additionally, these kind of modules often require redefining what a line means.

A Line in DCAF is a logical grouping of one or more channels and their data or name; an instance of 1 of whatever the module “does”. For example, a DAQmx module could define a line as a DAQmx line (say, AI1) and the channel it is to be mapped to. The J1939 module defines a line as a channel and the XNET signal.

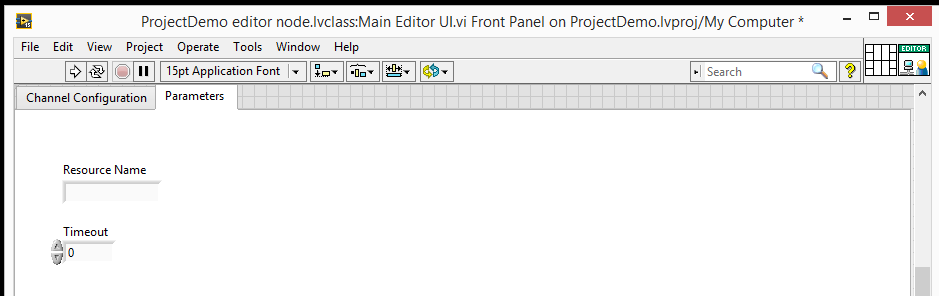


#### Defining Parameters

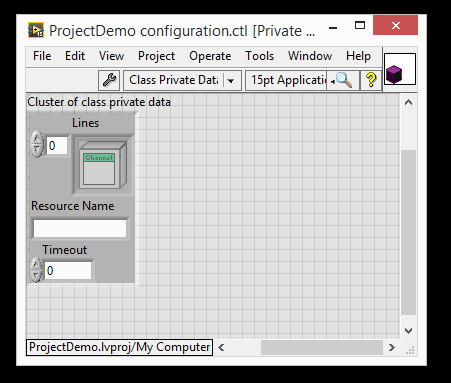
In a dynamic module, the Channel Configuration window allows users to input Channels and set their type through the editor. However, if the module needs extra, non-channel settings, then extra tabs would need to be added to the configuration UI VI.

Furthermore, while the Channel Configuration window has the options “Read from HW” and “Write to HW”, these may need to be customized for a specific application.

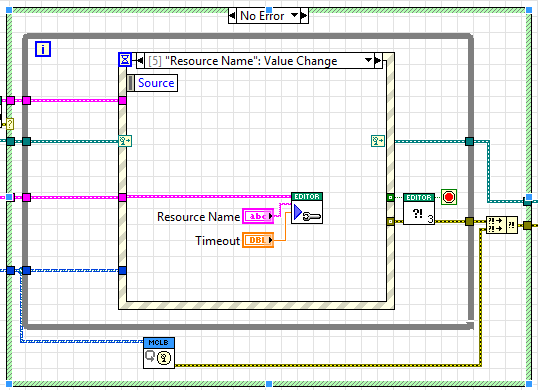
To add a set of parameters to the UI, go to the *Main editor UI.vi* contained within the *ProjectDemo editor node.lvclass*. Then add a page to the Tab Control and add controls to that new page

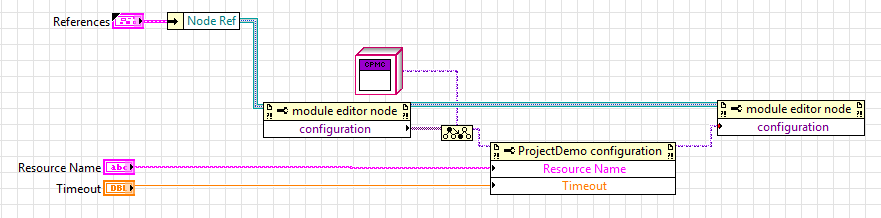


Because these were manually added, the editor itself will not automatically add the **Resource Name** and **Timeout** parameters to the configuration file on save. Similarly, it will not apply them to the *ProjectDemo configuration.lvclass*. First, add the necessary parameters to the *ProjectDemo Configuration.lvclass’* private data cluster and create accessors for the data:



Then, add the capacity for the editor node to write those values to the object. For example, this could be done by adding a new Event to the *Main Editor UI.vi’*s Event Structure to handle the event change. A new VI was created to handle this

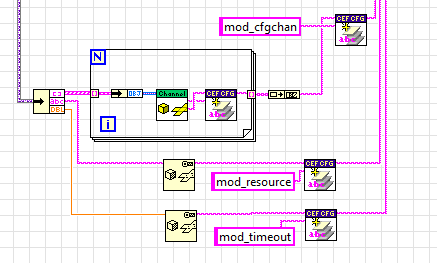




In here we are taking the references of the node itself and pulling out the configuration, modifying it, and writing that configuration to the node once more.

### To and From String

Most changes to the configuration class data also require updates to the configuration class’s To String and From String methods. These methods convert the data in the class object to a string and back and are required for saving and loading the class from a .pcfg file. When implementing these functions, it is often best to start with the To String method, and then implement the opposite behavior in From String. The most straightforward way to convert additional LabVIEW data into a string is to simply use the ‘Flatten to String’ function. Other types of flattening such as Flatten to JSON may have difficulty with special characters. LabVIEW classes should also generally be flattened manually with a special function as shown below.



The To and From string methods should also support the adaptation of previous versions of the configuration class. When making a change to the To String method, it is typically best to increment the version number. The From String method should then be able to correctly convert from the latest version of the To String as well as all previous versions. One of the best ways to do this is with a recursive call structure implemented in both the To String and From String methods that can migrate each version one at a time.

For example, here’s a To String implementation for a module whose latest config version is 3.0:

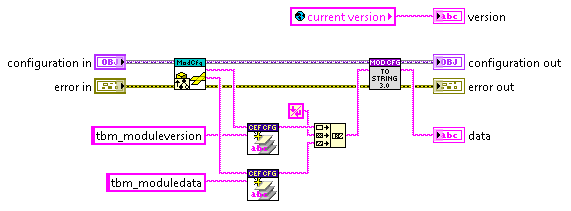


Figure : To String Top-Level

Opening the block diagram of the To String 3.0 VI reveals a call to version 2.0. When the 2.0 VI returns, 3.0 concatenates its data to the output string that 2.0 provides.

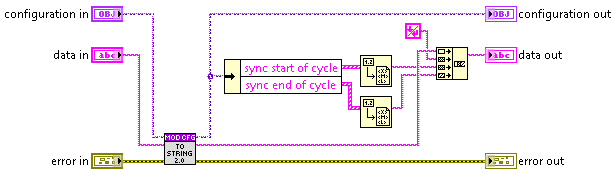


Figure : To String 3.0 Block Diagram

Looking inside of the To String 2.0 reveals a similar call to the previous version, 1.0. Again, it takes the string from the 1.0 call and concatenates its own data before passing the resulting string.

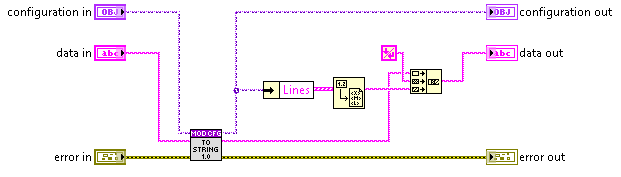


Figure : To String 2.0 Block Diagram

Since 1.0 is the oldest To String version in the chain, it simply adds its data to the string without making a call to another version first:

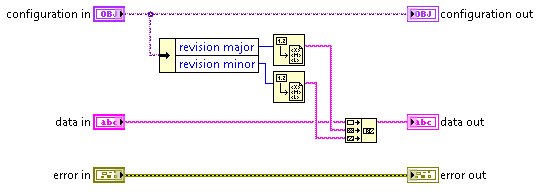


Figure : To String 1.0 Block Diagram

The *From String.vi* would need to be modified the same way.

Now that the values are part of the module configuration, they can be passed to the *ProjectDemo* *runtime.lvclass* if necessary, or used as part of the initialization.

#### Defining Lines

Lines tie the Channel information to what the module needs to do. After having done the necessary modifications to the Line.CTRL, the next changes need to happen within the Initialization code, so it will be treated in that section.

The editor needs a way add a line to the Channels tab. While it is possible that no changes onto the definition of a line are needed, if you must change a line (such that it creates two channels for example), this can be done through modifying the *Line Configuration Dialog* within the *Editor Node.lvclass*.

While DCAF will save the Line information into the configuration, the *init.vi* needs to have code within it to take the information in the line and use it to map some kind of data or physical device to the Channel.

### Customizing Behavior

Similar to the static module, the majority of the code that should be modified will be in Customize or Override folder. However, unlike in a static module, here the *ProjectDemo runtime.lvclass*’s init, input, process and output will be directly modified.

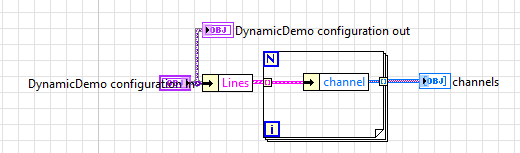
### Modifying the Initialization

Similarly to static modules, it is within the initialization that you would create sessions to hardware and other resources. Then you may pass the necessary resource to the Runtime class.

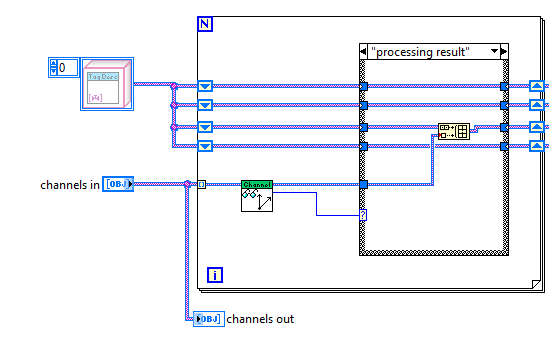
In Dynamic modules, the dictionary is passed to the different methods but there are no **scripted accessors** because it is not known what the channels will be or how many there will be. In order to access data from within the tag bus, the module developer will need to use the Table API in the input/output and either look up the channel by name in the specific method or to create a logical mapping between channel and index and pass that to the different methods through the runtime class (say, building an array of clusters of tag name and index).

Regardless of the method that will be used to access the table data, the init case first takes the channels, creates a dictionary, and passes that dictionary to the engine. This is all done through boilerplate code, explained here.

The *init.vi* is given a series of lines. It then pulls out all channels from the lines with the *Get Channels.vi*:



Then channels are separated into the different types (Input, Output, Processing Parameter and Processing Result) and a dictionary is created out of them. This is so that the engine only returns to the input, output and Processing modules the necessary tags and channels that it cares about for performance reasons and data encapsulation.



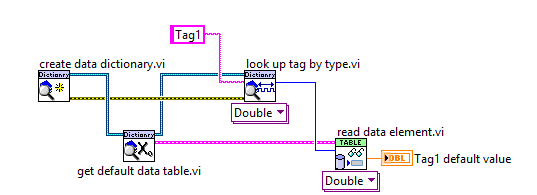
With the channels separated, a dictionary is created for each.



The dictionaries are used to read the information from the Tag Bus by index. From the Tag Bus Library

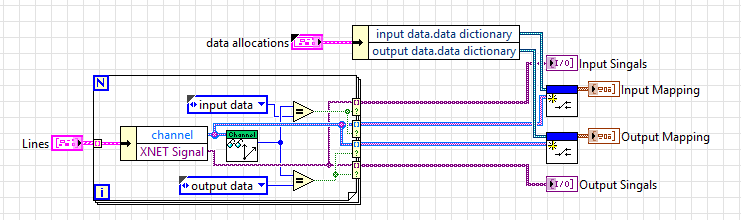
article ([found here](http://forums.ni.com/t5/Distributed-Control-Automation/LabVIEW-Tag-Bus-Library/gpm-p/3581721)), you use the Dictionary functions to get an index for a tag, and then use that index

to read the data from the Tag Bus: (image from Tag Bus Library article)

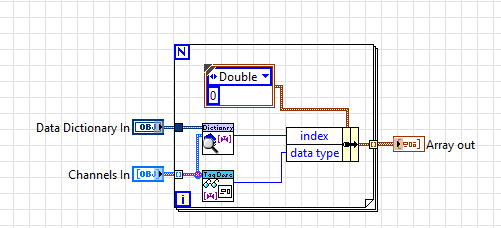


After the dictionary is created, there are two options for accessing data within the Tag Bus. Either the code can do the look up in the specific method such as the Input or Output, or the init can create a table of indices and tags. This is what the J1939 module does. The following images are taken from that module.

First, in the J1939 module, the Lines are pulled from the configuration class and passed to *Create Mappings.vi*. (Note that the term ‘Mapping’ here is a bit overloaded. In this case it refers to the mapping or link between the Module’s channels and its internal physical resources and not the mapping to the engine’s tags.) This VI takes the lines and separates the Inputs and Outputs, then maps the channels to the objects within the dictionary using the *Create Mapping.vi* (Notice that it is singular):



The *Create Mapping.vi* (Vis on the right) take the channels and searches for the appropriate tag:



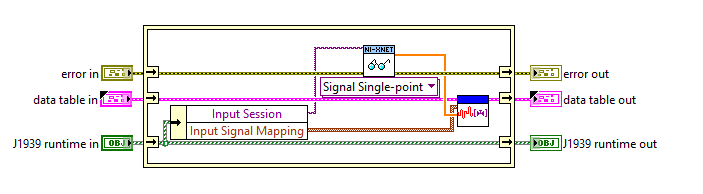
All of the mappings are passed into the Runtime class which contains the input and output mappings, as well as the hardware session.

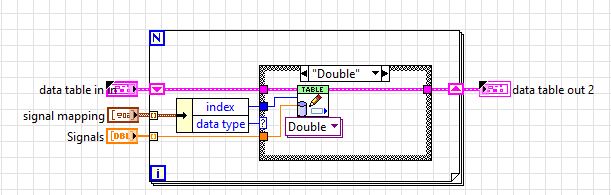
**Summary**: The init has boilerplate code to take the channels and create a dictionary. It is recommended to use the dictionary API to look through the dictionary for all channels and get an index for those channels. Then, pass the index and the channel information through the runtime so that the other methods can access channels without doing a look up. Use the indices from the init to write to the tag bus using the Table API. Using indices is the most performant way to access the data in the tag bus.

### Input, Processing, and Output Methods

Once the initialization has been written, the Input, Processing, and Output cases can be made. Note that specifically the *input.vi* and *output.vi* are what will need to be modified; recall that the Input.vi should be used to get a value and pass it to the tag bus (for example, by communicating with hardware), while the output should be from the tag bus and out. While the process is very similar to the Static Module case, there are no scripted accessors.

In the case of the J1939 module, the Runtime object here is unbundled and the session is used to acquire from hardware. Then, the *Convert signal to Tag Bus.vi* is called to take the input signal mapping and the data from the XNET read, and using the Tag Bus Table API to write the value to the data table for passing to the engine as Channels.





### Table API

The Table API is used to directly access the data on the dataset given from the module to the engine. This data set, refered to as the “tag table”, “data table” or “tag bus” is accessed by index. This index is obtained by doing a search on the dictionary in the *init.vi* case, as per the previous sections of this guide.

The most important pieces of the API are *write data element.vi* and *read data element.vi* which respectively write and read data in and out of the tag bus. It is also possible to read an array of elements (*read multiple elements.vi)* or write them (*write multiple elements.vi*).

The API is polymorphic and acts upon one data type at a time, which is why the J1939 module uses it within a case structure and calls a specific type. Using a class-based approach was considered, but ultimately disregarded for this because it would be add overhead and lower performance.

It is also possible to divide data in different groups, and move the groups between tag buses. This is covered within the [Tag Bus library](http://forums.ni.com/t5/Distributed-Control-Automation/LabVIEW-Tag-Bus-Library/gpm-p/3581721) article.

## Module Testing

As modules are developed in three pieces, it is often best to test each piece individually before putting them all together. The first piece to test should typically be the configuration class. This class should generally be possible to test with Unit Testing. As part of the template, a *ProjectName tests.lvlib* is created with the modules to demonstrate a few of the tests that could be done. These must be run manually, but can be automated using the LabVIEW Unit Test Framework. This approach is highly recommended and more details can be [found here](http://www.ni.com/white-paper/8082/en/).

Many of the Vis in the testing library depend on *Create Test Configuration.vi*, which creates a default configuration object with a set of channels for all the supported types, and adds those channels as lines through the *Add Line.vi*. It is in the *Create Test Configuration.vi* that you may need to add logic such that lines are created correctly – for example, if a line is 2 channels and a string in a custom module the bundle by name would need to be modified to allow for this. The J1939 module, for example, populates the XNet Signals in the channels as well as the XNET Settings property of the class.

At a minimum, it is important to test the To String and From String functionality of the configuration class before proceeding. If this functionality does not work, then the module may not behave properly with configuration editor.

Once the configuration class has been tested, either the runtime or the editor node class can come next. For these classes, an approach of unit testing and manual testing is often required.

For the runtime class, unit testing is also the best approach to take when possible. One simple test to run is the *init runtime test.vi* which creates a test configuration, initializes the module and checks that the channels in the module can be found in the Data Dictionary. Even though the module methods are designed to run within an engine, it is still possible to create a test harness VI that passes in values and reads results from each runtime class method. Unlike the configuration class, the runtime class often has hardware dependencies that prevent simple testing on Windows, but these can still be overcome. Start with something simple, and add more unit tests as needed.

If performance of the runtime code is important, standard benchmarking techniques such as the use of the Real-Time Execution Trace (RTET) toolkit can be helpful. Once again it is often easier to use these tools with simple test benches before running the module within an engine.

For the editor node class, once again start by testing as much functionality as possible with unit testing. Manual testing will likely be required for most of the user interface interactions. This testing is typically best done directly in the Configuration Editor. Edit the search paths of the editor to ensure that only one copy of the module is loaded from disk from the correct location. Once loaded in the editor and added to a configuration, DCAF enables debugging of a module UI by allowing a user to open the block diagram of a plugin by right-clicking in empty space (above any tab controls) and selecting **Open Block Diagram**. This will take you to the diagram of the running Main Editor UI, where regular LabVIEW debugging techniques can be used to troubleshoot the module. This is mostly important for dynamic modules because modifying the UI is not expected of static modules.

After testing each component of a module individually, build out a more complex integration test using the DCAF configuration editor and execute the module within an engine. The Engine’s benchmarking capabilities can also be used to validate the performance of the module.