Configuration Editor Framework Introduction

# Overview

Application configuration is a large task for application architecture. In an attempt to make system configuration easier for designers as well as end-users, configuration editors are often implemented.

In this document we cover the Configurator Editor Framework which is a starting point for creating a custom configuration editor in the LabVIEW Development Environment. The framework consists of a tree control and subpanel API, 2 LabVIEW project template, and 2 abstract LabVIEW classes.

The example configuration editor described in this paper is for creating a configuration tool for an embedded data logging system, but the framework can be used for creating other configuration editors for automated test, machine control, or other applications.

In this document we cover the use of the basic template and the core concepts of the framework. The advance project is covered in the Configuration Editor Framework Advance Project Template document.

# Requirements

* LabVIEW 2013 Development Environment or later
* [Asynchronous Message Communication (AMC)](http://zone.ni.com/devzone/cda/epd/p/id/6091) Reference Library

# Introduction

System Configuration is used to store and access semi-static data in an application which describes the hardware and behavior of the application. Configuration decouples the need to edit source code when a user needs to:

* make changes to parameters
* change hardware specific information
* configure application specific behavior
* customize applications for a specific use

The data for a configuration is stored in a file or database. At run-time, applications open the data from a file or database, rather than having the developer hardcode the information into the source. Time and money is saved by removing the need to validate new source code.

When editing configuration the user can work directly on the file with a text editor, excel or other file editing programs. General purpose file editors can work for simple applications, but forcing users to use these non-specific tools makes configuration hard to understand and error prone.

Configuration editors are typically graphical user interfaces (GUIs) that enable a user to create/edit configurations so that the resulting file conforms to a format that an application can interpret. By enabling and limiting the interactions of data in a configuration the user can create a scalable and maintainable way to store information. Using a configuration editor provides several advantages:

* Information can be input in different ways
* Invalid configurations can be restricted by the configuration UI

Decoupling the programmatic requirements from application requirements, makes it easier for users to visualize a complex system

* Version managing can be done by the Configurator
* Multiple file formats can be supported
* Easier to configure complex system with large number of configuration parameters.

# Common Features

National Instruments products include many examples of configuration editors. Measurement and Automation Explorer (MAX) is a configuration editor commonly used to configure NI Hardware. Figure 1 shows the GUI for MAX.

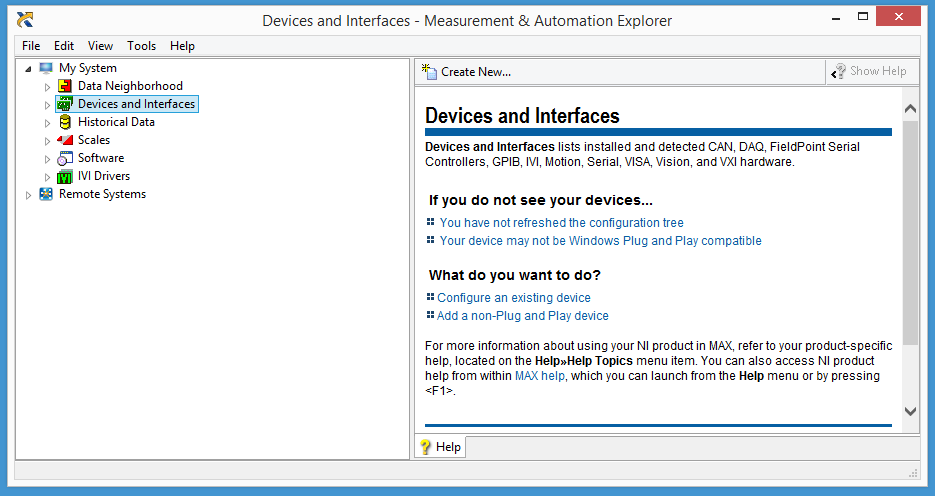
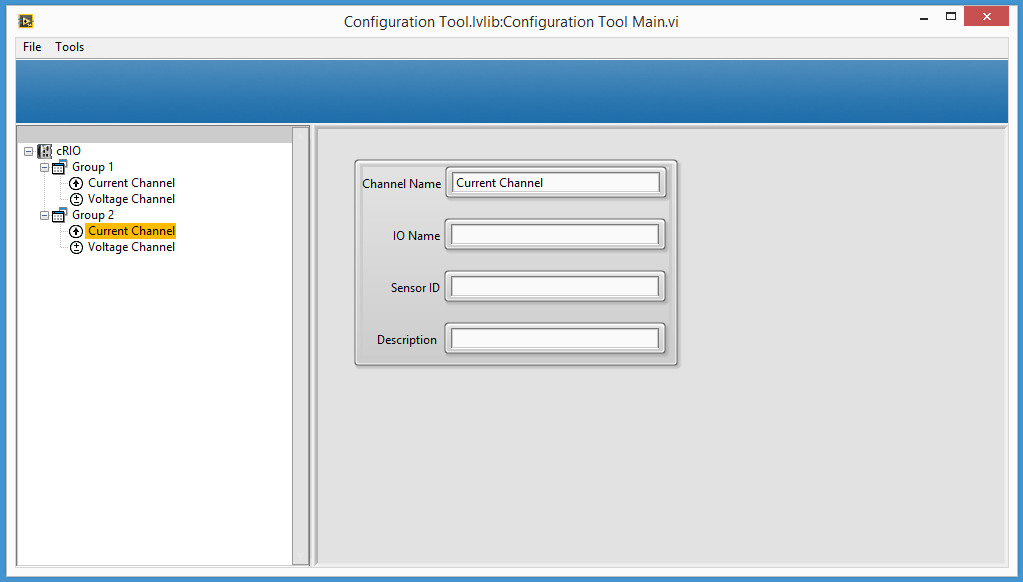


Figure 1 NI MAX Graphical User Interface

Another configuration editor is the LabVIEW Project Explorer. The Project Explorer saves the information as an XML file with the extension “.lvproj”. Even LabVIEW has a configuration file called “labview.ini” which is stored as human readable initialization file which can be edited through the options dialog.

After reviewing the above configuration editors, four common elements can be extrapolated.

* There is a graphical way to navigate through the configuration. Typically this is using a tree control.
* After we navigate to a node in our tree data structure, we can edit the data stored at that node. Typically this is through another window (floating or attached).
* Nodes in the tree data structure can be added, duplicated, removed/deleted, and moved to another location in the tree data structure.
* There are multiple methods of setting the configuration (single element, multiple element or importing an external configuration)



Configuration

Navigation

Figure 2 Common Layout for a Configuration Editor

# Implementation

The ultimate goal of this framework is to provide the majority of features required to make a configuration editor while still leaving enough flexibility to customize the editor for specific applications. The framework accomplishes this goal by providing a tree control API in LabVIEW as well as an easy way to have different views of the same data. Although the framework promotes a tree control as main way to visualize the configuration, custom views can be created.

The central part of the Configuration Editor Framework is the repository. The repository is the location where all configuration parameters are stored while editing. Interaction with the repository occurs through the tree, the configuration file, and any custom views developed. The tree is a tree control and tree API. The configuration file is used to load and save parameters to the repository, and is most likely what our final application will be using for configuration. Custom views provide difference ways to visualize and interact with the repository separate from the tree. An example would be a multiple variable editor view where all information is flattened to a single table for easy filtering. In order to pass data between the different views, configuration data must pass through the repository.

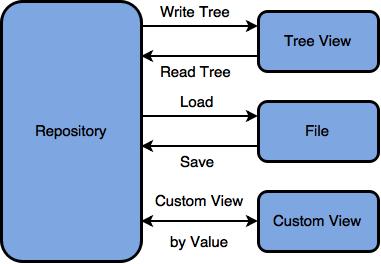


Figure 3 CEF Repository and Views relationship

## Project Templates

When the CEF is installed 2 project templates will be added to the new project wizard, The Configuration Editor Template and the Hierarchical Configuration Editor template.

The configuration editor provides all the basic features of the CEF, and provides is the most flexible option but requires more programing.

The Hierarchical configuration template has some of the classes defined and implemented for the user. This includes the repository class, the tree node, and the file operations. Having this allows for creating projects faster but limits the flexibility. Also this project doesn’t not includes the custom view implementation. Most applications will benefit of the additional code in this template.

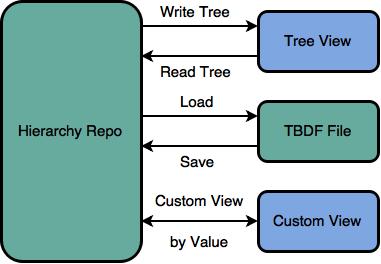
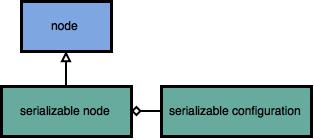
 

Figure 4 Hierarchical Template implemented classes

## Repository Class

The repository (repo) is where configuration information is stored in the editor. The parent class contains an array of paths and names that can be used to get the file path of a class from the class name. In addition the repo might contain additional information and auxiliary data structures for the different views. The repo is implemented as an abstract class and all its methods are expected to be overridden by a child class that the developer creates. The following are the provided class methods. Although the framework is open source, most developers will not need to know about the supporting subVIs.

|  |  |
| --- | --- |
| VI | Description |
| Open | Loads the saved information of the repo. It could load multiple files at the same time. An option to add file versioning could be included in the files, and the code to manage changes would be in this VI. |
| ReadTree | Transfers information between the repo and the tree control nodes. This should be called before custom views are launched or saving to disk. |
| Save | Stores repo information to file. It could save in multiple files or formats at the same time. File versioning is suggested. |
| Validate | An abstract method for validating the information inside of a repo is accurate. This method should be called before saving the information to file. |
| WriteTree | Transfers information between the repo and the tree control. This method should be called after the custom views are called or saving to disk. |
| Initialize | Initializes the repo. StoreClassPath method should be called in this VI |
| CustomView | Calls the custom view specified in the view name string. The purpose of this method is to allow adding a Configurator editor without modifying the main VI. |
| GetClassPath | Gets the path to the Class based on the Class Name.  The default behavior is build the path of the class on top of the class folder path. The name of the class and the folder it contains needs to be the same. A simple lookup from index and name would be the other option. |
| StoreClassPath | Stores the paths and classes name that the repo use.  The default behavior is to store the path to the main folder containing the classes and expect all classes to be in folders with the same name as the class. If a more complex file structure is required this file needs to find all the classes and store the name and path pairs. |
| Validate | Placeholder for creating system wide validations. |
| GenerateMenu | This Vi is called when creating a new repo, and creates the file menu for the configuration editor |
| HandleMenu | This vi should be overridden if any additional menus were added in the GenerateMenu.vi |

Table 1 Repository.lvclass Methods

# Serializable Configuration class

## This is an abstract class created to store information for specific nodes, and is used in the hierarchical template.

|  |  |
| --- | --- |
| VI | Description |
| to string | Flattens to a String. |
| from string | Unflattens from a string. |
| create default alias | This is the display name that is default in the tree for the hierarchy node. The parent implementation is to use the class name. |
| create new identifier | Called when initialized creates a new UID (unique identifier). |
| destroy | This Vi is for cleaning up references if used inside of the class. |
| duplicate | This will call when duplicating a class. |
| validate | Validates the configuration. |
| execution class name | This is used when the configuration class has a runtime component. Having this VI allows that the same configuration class can be used with different runtime classes. |
| dynamic initialize | This the portion of initialize that can be overridden. The initialize method contains methods that must be executed every time and should not be changed. |
| instance count allowed | Tells how many instances of the class are allowed on the system. |
| refresh and repair | This is called by the right click menu on the tree with the option repair. It is called inside the Attempt Repair.vi |
| callback on load | This VI should be called when loading the class for example in the FromRepo.vi |

Table 2 Serializable Configuration.lvclass Methods

## Hierarchy Repo Class

This class inherits from the repository class. The only Vis that should be overridden from this class are the ones that are also part of the repository class. And in most cases there should not be any need to create child classes for this class. To acces and interact with this class an API has been created shown in Figure 6 Hierarchical Repository API

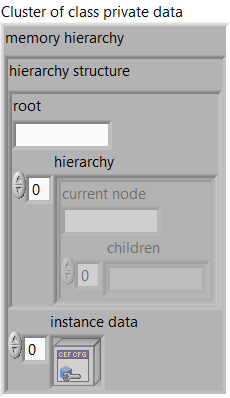


Figure 5 Hierarchy Repo Private Data

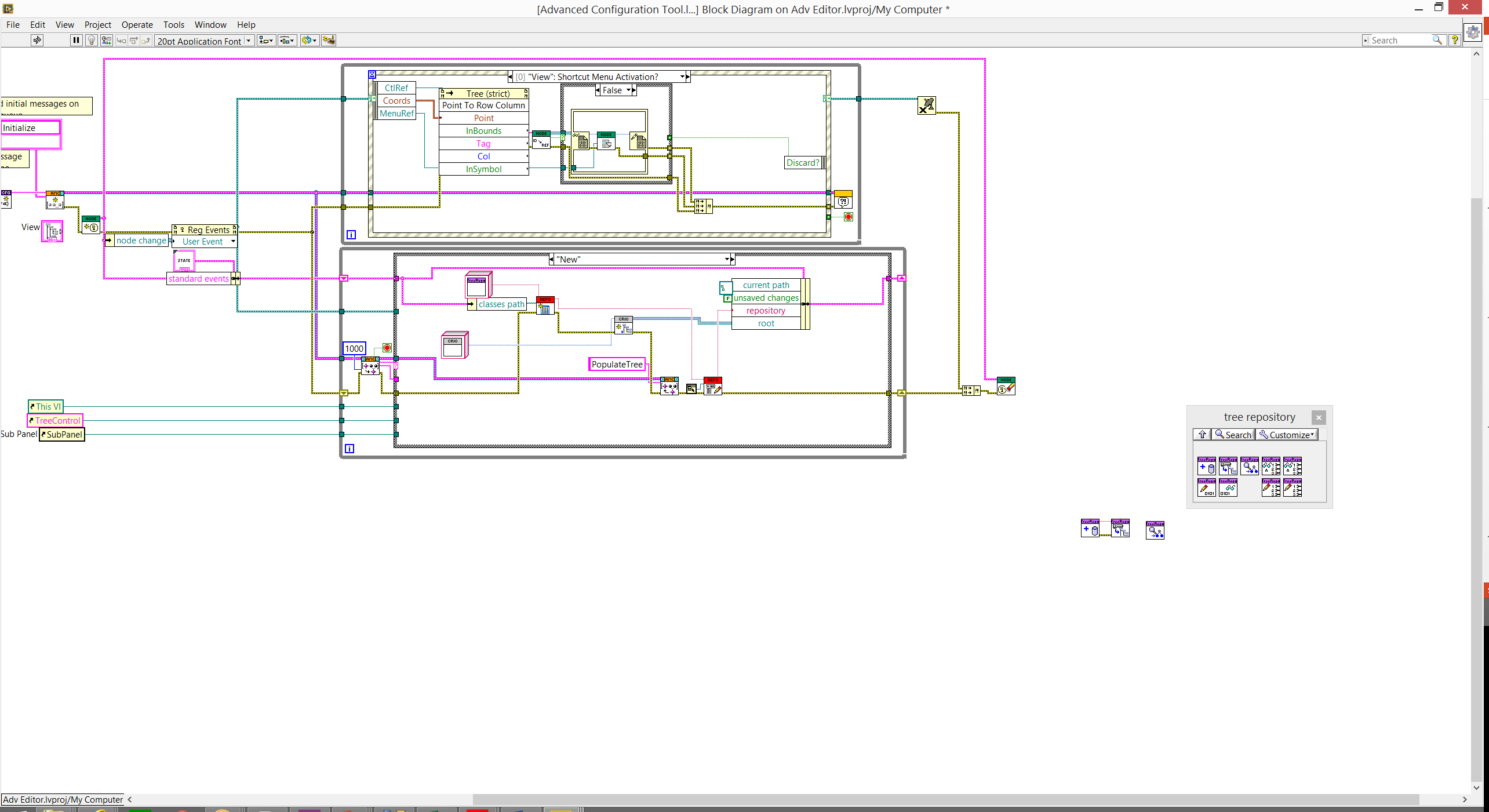


Figure 6 Hierarchical Repository API

## Node

The concept of a node is implemented as an abstract class. As with the repo class, developers are expected to create classes that inherit from it, for every node type their configuration editor. Every leaf of the tree will be represented by a node. If multiple leaves will have too much functionality in common an abstract class should be created.

Each node is accessed by reference using a Data Value Reference (DVR) and stores the reference to its parent node and all its children. The tree control stores this reference as a string.

|  |  |
| --- | --- |
| VI | Description |
| Duplicate | Implements the specific behavior of a node when it is being duplicated  Can be used to change the name of the new node for example adding Copy to it. |
| ToRepo | Transfers the information from the tree node into the repo. How it is stored in the repo depends on the Repo. |
| FromRepo | Grabs information from the Repo and stores it in the nodes. |
| UIRef | Stores the static reference to the UI of the node. |
| Initialize | Initializes the data of a node to its default state.  The object is not created inside of this VI, because it is created from a File depending on user selection. |
| ShortcutMenu | Defines the behavior of the shortcut menu of the node. |
| EditOptions | Contains the options for the Menu that will be displayed when doing right click on an item.  The sub nodes class configures which nodes can be added to this nodes  And the edit options determine which options are available for the node.  This configuration is stored in this VI in case some dynamic configuration is required for the menu options or sub nodes. |
| GetText | Returns the text representation of the node used for the tree control. |
| LinkNodes | Links 2 nodes together one as a parent and one as a child. |
| UnlinkNodes | Breaks the link between a child and a parent node. Removing the reference to each other in both nodes. |
| NodeUI | UI that will be called when the node is selected. |
| InnerCallbackOnLoadUI | There is a static VI that is callback on Load UI, by doing this it will allow to add extra functionality when loading the UI. |
| GetOverlay | Specifies which pre-loaded overlay to add to the Glyph. |
| AddGlyphtoGManager | Optional Override that adds any necessary glyphs to the editor. The parent implementation searches the local directory for available .png files. |
| RemoveChildfromNode | This vi runs inside of the UnLinkNodes and allows complex functionality. |
| AddChildtoNode | This vi runs inside of the LinkNodes and allows complex functionality. |
| MenuHandler | This should handle custom menus for the Node. |
| CleanUpNode | Is the dynamic dispatch of the RemoveNode. |

Table 4 Node.lvclass Methods

# Serializable Node

This class inherits from Node and contains a serializable configuration inside. The FromRepo and ToRepo classes are already implemented. In most cases the only overrides that will be required when using this node will be the UIRef, create a new UI and the GetText VIs.

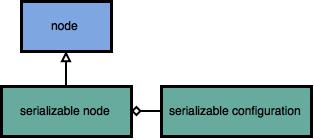


Figure 7 Serializable Node

# Getting Started Configuration Editor Template:

This section explains how the example classes were designed and created using the Configuration editor template. The classes described in this section of the document are included when creating a new project as a reference. In your application they can be removed or modified freely.

In this example we have a simple configuration file which consists of an array of channels stored in XML format. This same data can be represented multiple ways. The editor allows us to have 4 different representations of the configuration data. This is shown on images Figure 8, Figure 9, Figure 10, and Figure 11

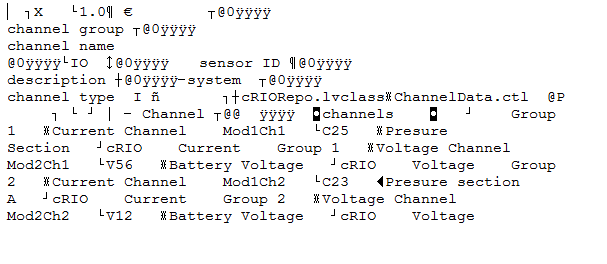


Figure 8 File Representation

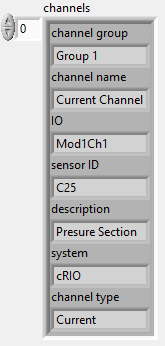


Figure 9 LabVIEW data

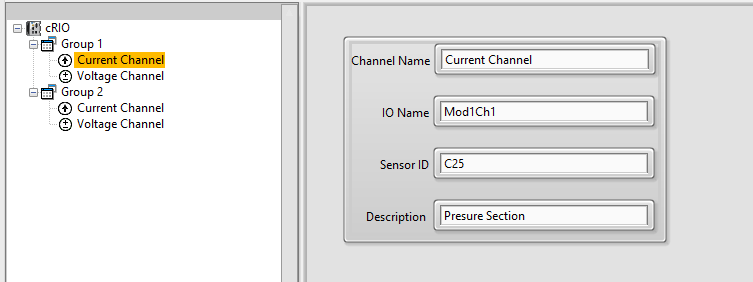


Figure 10 GUI Representation

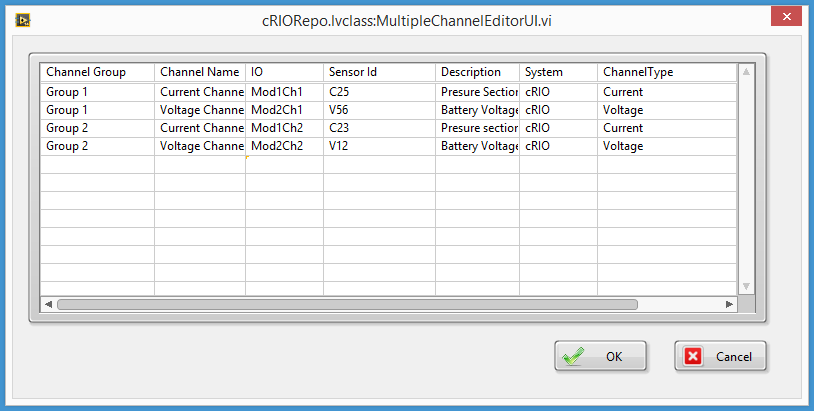


Figure 11 Multiple Channel Editor Representation (custom view)

### Creating an new project

The Configuration Editor Framework is distributed as a sample project.

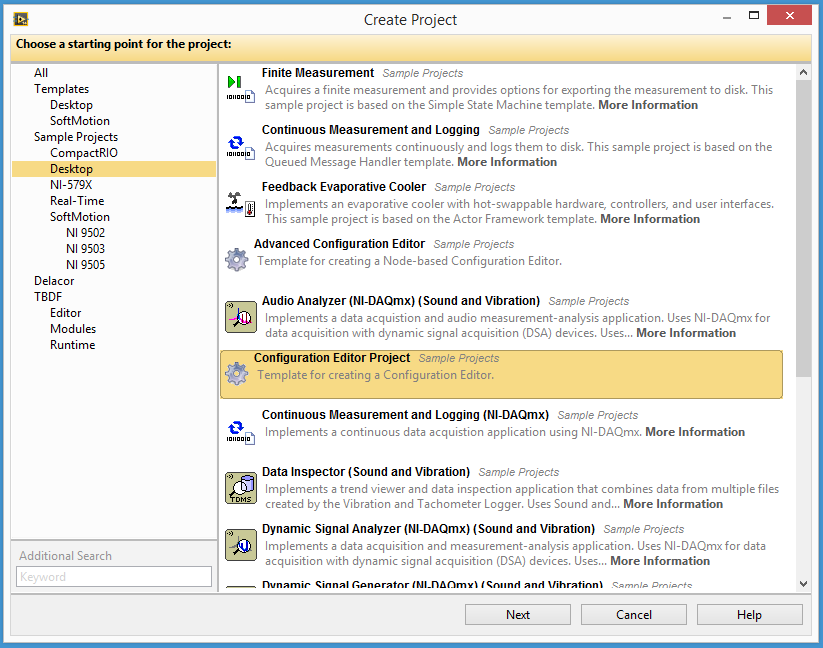


Figure 12 Create Project from template wizard

The following project will be generated:

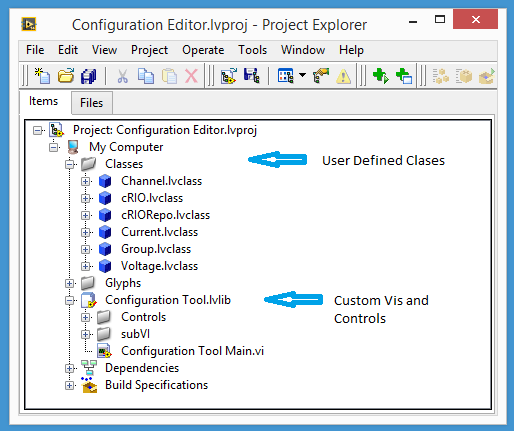


Figure 13 Configuration Editor project

### Creating a Repository

The repo is the central place for your data and it allows it to travel between views, in general a simple repo will contain the cluster or LabVIEW representation of your data. In addition you want to add all the other auxiliary data structures that will be useful for your repo.

In the this example we created the cRIOrepo.lvclass which contains two elements, the array of channel clusters we will save to disk, and a placeholder temporary channel that would be used to store information when sending information to repo.

In the case of the cRIO repo we override the following methods:

* Open: Here we are specifying to read the configuration from a XML file verify it’s the right version of the configuration cluster if not it generates an error.
* Save: Saves the repo information into the XML file.
* Write Tree: Creates a cRIO node object to be used as root and calls the from ToRepo method of the node.
* CustomVIEW: Calls the MultipleVariableEditorUI.vi that is the custom view created for this configuration editor

In addition to these VIs, data access VIs were made so that it is possible to access the information inside the repository to the rest of the application.

1) Create the new class

a) Right click on the class’s folder and select new class

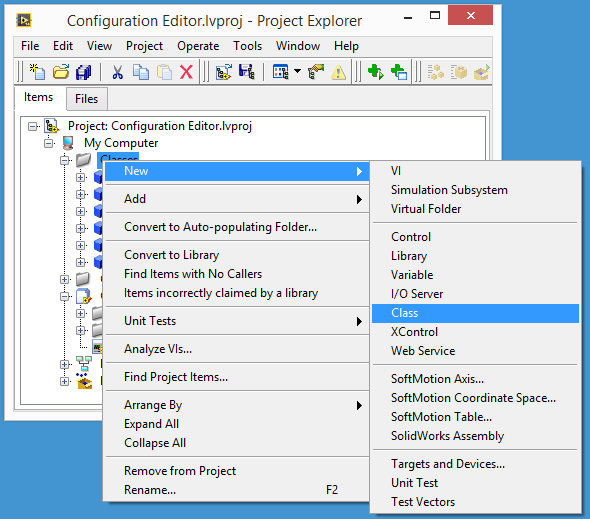


Figure 14 creating new class

2) Name your class and save it to disk.

3) Inherit from Repository.lvclass

a) Select class properties

b) Select Change Inheritance

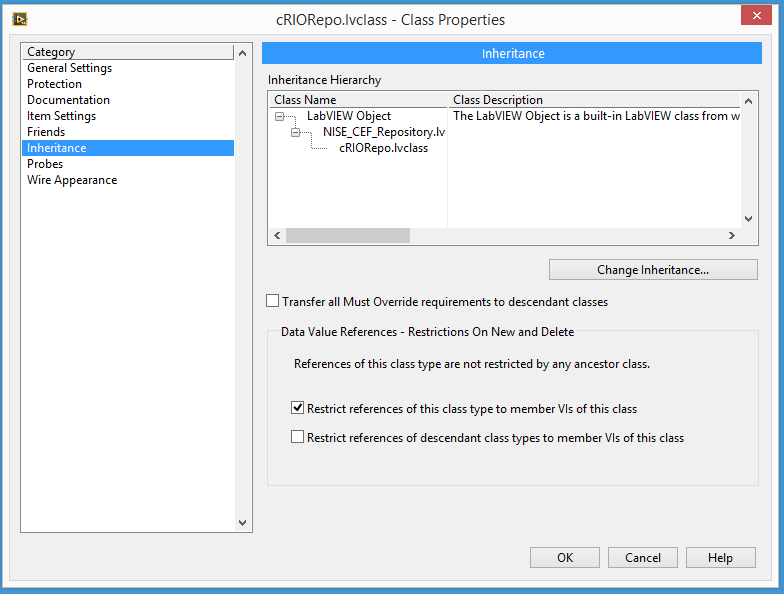


Figure 15 changing inheritance

c) Inherit from Repository.lvclass

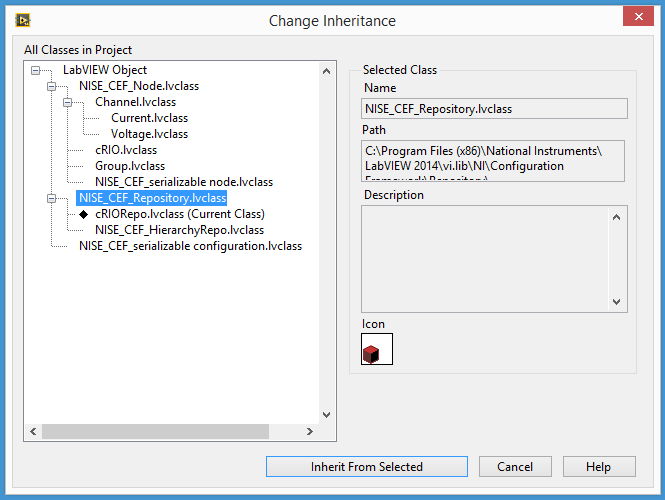


Figure 16 selecting new inheritance

### Creating Tree nodes

When we are working on the tree control, each different element on the tree needs to be represented by a class. All classes should inherit from the Node.lvclass.

The methods that will be overridden by vary for each class. In this example we need to create cRIO.lvclass that will be our root node, group.lvclass, voltage.lvclass and current.lvclass. Because both voltage and current are channels and have several properties in common it makes sense to create an abstract parent class for both channels called channels.lvclass that will be inherited by both voltage and current. This also makes it easier to add additional channel types in the future

The class hierarchy is show In the following picture

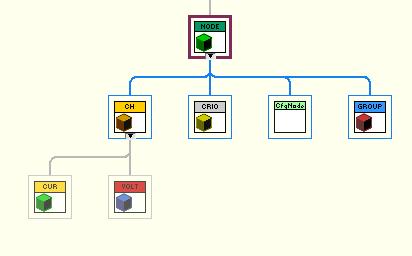


Figure 17 node class caption

cRIO.lvclass

This node will act as the root node in the Configurator.

* Initialize: Sets the system name to the default value cRIO.
* EditOptions: Allow to add sub nodes with of the group class. Because it is used as the root it cannot be removed, duplicated or dragged.
* ToRepo: Stores the system name in the auxiliary channel data and initializes the channel array.
* FromRepo: All channels should belong to this system. Gets the system name from one of them and search for the name of the different groups in the array.
* GetText: Sets the glyph to cRIO glyph and uses the group name as the text.
* UIRef: Sets the reference of the UI to cRIOUI.

Group.lvclass

This node can contain multiple channels of all both channel types and the only property it has is the group name.

* Initialize: Sets the Group name to the default value (Group).
* Duplicate: Adds the word copy to a group name that had been duplicated.
* EditOptions: Allow to add sub nodes consisting of current and voltage Channels. The group can be removed and duplicated, but can’t be dragged.
* FromRepo: In this VI the channel array is scanned to search for all the channels that should belong to the group.
* ToRepo: Stores the group name in the auxiliary channel data.
* GetText: Sets the glyph to group glyph and uses the group name as the text.
* UIRef: Sets the reference of the UI to GroupUI.

Channel abstract class

This is an abstract class and is never display on the tree control. It defines most of the methods for the channels than inherit from it.

* Channel UI and Channel UI Ref. Defines the UI that will be used for all channels. If a specific channel needs a different UI the UI ref can be overridden.
* EditOptions: Allows the channels to be moved, removed and duplicated. And specify no sub nodes can be added to them.
* GetText: Sets channel name as the test and a default Glyph is used in case a channel didn’t specify it.
* ToRepo: The complete channel information is populated using the values of System and group from the current channel auxiliary data, and writes the information into the main Array Channels.
* FromRepo: Using the index stored in the from repo cluster, gets all the channel info on that index in the array.

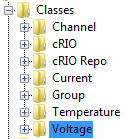
There are two data access method VIs that allows you to get the data in the VIEW cluster format or in the Channel cluster format.

Current channel and voltage channel

These channels inherit most of their methods from the Channel abstract class. The only specific VIs for these classes are Initialize and GetText.

* Initialize: Sets the channel name and the channel type to the specific ones of the corresponding classes.
* GetText: Sets the glyph to the right value and use the text method from the parent (channel).

1. Define where the classes will be stored the default path is in the /project folder/Classes. The Configurator expects all classes to be stored in a single folder containing the folder for all the classes.



NOTE: The path to the main classes folder is specified when initializing the repo and classes should be stored in a folder that has the same name that the class. If your classes don’t have this order in disk the GetClassPath and StoreClassPath methods from repo need to be overwritten.

1. Create the new class
2. Save to disk.
3. Change inheritance to inherit from Node.lvclass.

### Creating the custom view

It is often more convenient to edit multiple channels at the same time, because of this we created a custom view that is a multiple parameter editor.

For creating the multiple parameter editor we override customview.vi to call multipleparameterUI.vi

If more than one custom view is required you can pass the name thorough the view name string and use a case structure to select between the different options

# Getting Started Hierarchical Configuration Editor Template:

This section explains how the example classes were designed and created using the Hierarchical Configuration editor template. The classes described in this section of the document are included when creating a new project as a reference. In your application they can be removed or modified. This example does not include the custom view as it is covered on the main template section.

### Creating an new project

The Hierarchical Configuration Editor Framework is distributed as a sample project.

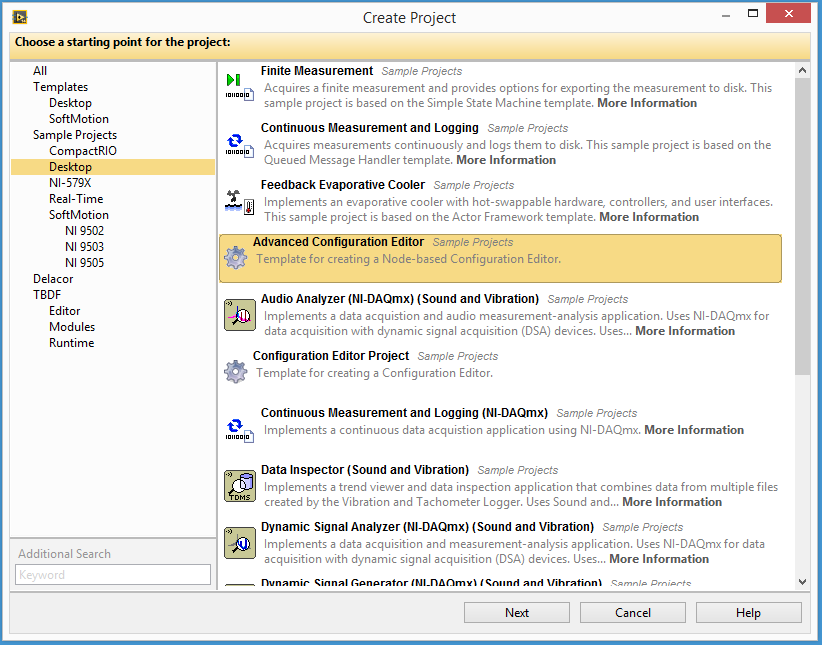


Figure 18 Create Project from template wizard

The following project will be generated:

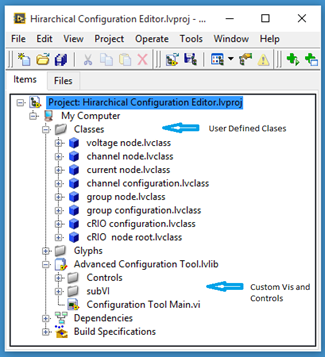


Figure 19 Hierarchical configuration Editor project

### Repository

With the Hierarchical project there is no need to make any changes to the repository in most situations. In case some changes need to be done, the procedure would be the same creating a repository on the Configuration Editor framework but it should inherit from NISE\_CEF\_HierarchyREpo.lvlcass.

### Creating Tree nodes

When we are working on the tree control, each different element on the tree needs to be represented by a class. All classes should inherit from the Node.lvclass. Also every node should have a configuration class. But the same configuration class can be used for similar nodes.

The methods that will be overridden by vary for each class. In this example we need to create cRIO.lvclass that will be our root node, group.lvclass, voltage.lvclass and current.lvclass. Because both voltage and current are channels and have several properties in common it makes sense to create an abstract parent class for both channels called channels.lvclass that will be inherited by both voltage and current. This also makes it easier to add additional channel types in the future. Also we will need to create the configuration class for group, cRIO and channel.

The class hierarchy is show In the following picture

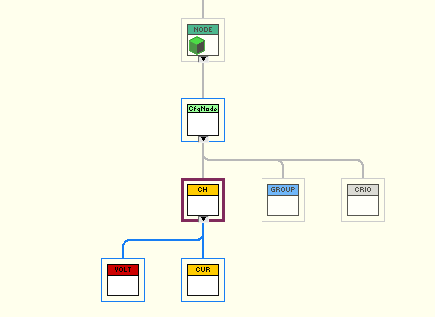


Figure 20 Node Class Hierarchy

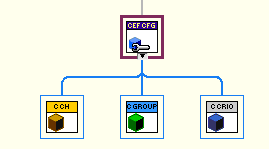


Figure 21 Configuration class Hierachy

cRIO node root.lvclass

This node will act as the root node in the Configurator.

* Initialize: Sets the system name to the default value cRIO. It is stored inside of the cRIO configuration class
* EditOptions: Allow to add sub nodes with of the group class. Because it is used as the root it cannot be removed, duplicated or dragged.
* GetText: Sets the glyph to cRIO glyph and uses the group name as the text.
* UIRef: Sets the reference of the UI to cRIOUI.

cRIO configuration.lvclass

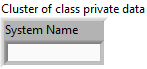


Figure 22 cRIO configuration private data

Only properties we are interested at cRIO level is the system name. The System name field is created in its private data and methods to access this information.

group node.lvclass

This node can contain multiple channels of all both channel types and the only property it has is the group name.

* Initialize: Sets the Group name to the default value (Group).
* Duplicate: Adds the word copy to a group name that had been duplicated.
* EditOptions: Allow to add sub nodes consisting of current and voltage Channels. The group can be removed and duplicated, but can’t be dragged.
* GetText: Sets the glyph to group glyph and uses the group name as the text.
* UIRef: Sets the reference of the UI to GroupUI.

group configuration.lvcass

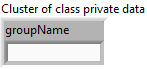


Figure 23 group configuration private data

Only properties we are interested at cRIO level is the system name. The System name field is created in its private data and methods to access this information.

Channel node.lvcass (abstract)

This is an abstract class and is never display on the tree control. It defines most of the methods for the channels than inherit from it.

* Channel UI and Channel UI Ref. Defines the UI that will be used for all channels. If a specific channel needs a different UI the UI ref can be overridden.
* EditOptions: Allows the channels to be moved, removed and duplicated. And specify no sub nodes can be added to them.
* GetText: Sets channel name as the test and a default Glyph is used in case a channel didn’t specify it.

There are two data access method VIs that allows you to get the data in the VIEW cluster format or in the Channel cluster format.

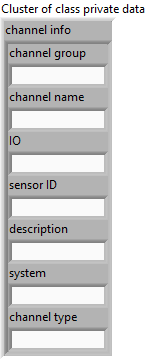


Figure 24 channel configuration private data

channel configuration

In here we created the fields for all the relevant information for channel as well as access methods to this information. In addition there are overrides in the following methods

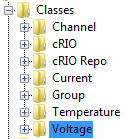
* validate: this Vi makes a simple validation making sure the sensor ID is not empty
* from string: This VI includes code for version management of the channel configuration.
* to string: This vi flattens the data in the configuration cluster and adds versioning information.

Current channel and voltage channel

These channels inherit most of their methods from the Channel abstract class. The only specific VIs for these classes are Initialize and GetText.

* Initialize: Sets the channel name and the channel type to the specific ones of the corresponding classes.
* GetText: Sets the glyph to the right value and use the text method from the parent (channel).

1. Define where the classes will be stored the default path is in the /project folder/Classes. The Configurator expects all classes to be stored in a single folder containing the folder for all the classes.



NOTE: The path to the main classes folder is specified when initializing the repo and classes should be stored in a folder that has the same name that the class. If your classes don’t have this order in disk the GetClassPath and StoreClassPath methods from repo need to be overwritten.

1. Create the new class
2. Save to disk.
3. Change inheritance to inherit from Node.lvclass.

## Feedback

This reference application was created by the NI Systems Engineering group.

We **do not** regularly monitor Reader Comments posted on this page.

Please submit your feedback in the [Configuration Editor discussion forum](http://forums.ni.com/ni/board/message?board.id=Components&thread.id=241) so that we can improve this component for future applications.

Please direct support questions to [NI Technical Support](http://www.ni.com/support/). Discussion