# Creating a New Application – Outline and Checklist

1. Create a Main VI for each execution target (windows, cRIO, etc.) using DCAF's execution templates.
2. Download existing DCAF module plugins from the community and design, build, and test new modules if needed (See Ch. 4).
3. Configure system behavior using the DCAF Standard Configuration Editor.
4. Load the code for each dependent plugin module within the appropriate Main VI.
5. Deploy an up-to-date config file to each execution target at the appropriate location.
6. Run the application from the development environment and troubleshoot as needed.
7. Build the application into an executable and troubleshoot as needed once again.

## Creating a Main VI Using Execution Templates

DCAF is distributed as a collection of APIs and not as a pre-built executable. Therefore, running a DCAF configuration requires the creation of a top-level application VI that reads the configuration and executes it. This is typically done in the Main VI of an application, and creation of this initial VI is simplified by two execution templates included with the installation of DCAF – the DCAF Basic Execution Template and the DCAF Execution Service.

Before using these templates, first determine how many unique Main VIs will be required for the full application. It is common for each execution target in an application to have its own Main VI, but when possible it is better to prevent code duplication and reuse the same Main VI for multiple execution targets. If changes between system behavior can be localized to changes in DCAF configurations, the same top-level VI can be reused.

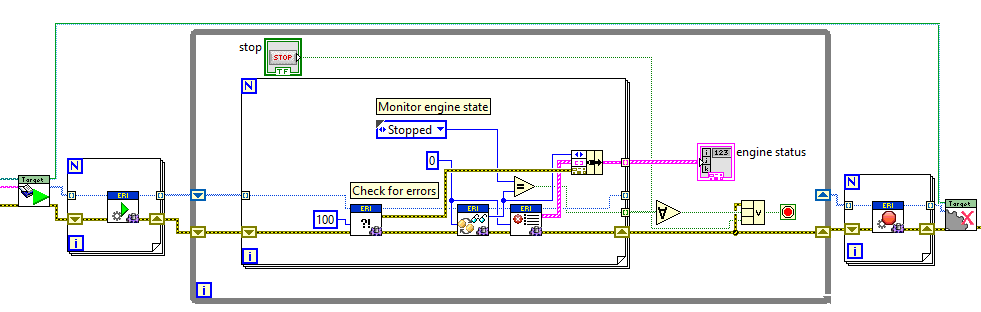
After determining which execution targets will use which Main VIs, decide which application template would be the best to start from when building each top-level program. The Basic Execution template is a simple program that launches one or more engines, checks for errors, and can stop the engine with input from the UI. This template is typically best suited for User Interface applications that can shutdown on error.

The Execution Service template is a demonstration of how to launch DCAF as a background service. This allows commands to the engine to start, stop, reinitialize, etc. to come from multiple threads within the application. The application also continues to run on error and demonstrates DCAF's ability to reload a program without stopping the main program. This template is typically a better starting point for targets that need to run headlessly.

Both templates are simply starting points for an application. Users should expect to modify these templates to suit the specific needs of their application. However in cases where all application functionality is contained within DCAF modules, these templates may be sufficient to use 'as-is'.

### DCAF Basic Execution Template

This template creates a project with two VIs, a *Host Main.vi* and a *Host Module Includes.vi*. These VIs may need to be moved to the hardware that will run the DCAF system – for example, you may need to use the Project to add a cRIO, then drag the *Host Main.vi* and *Host Module Includes.vi* to the cRIO before continuing.



The *Host Main.vi* first loads the configurations with the *load and initialize target with engines.vi* and outputs an array of the engines in the target, as well as a reference to the target itself. Notice that the target runtime reference is simply closed later. The array of runtime engine interfaces is passed to a For Loop that indexes through the engines and starts them sequentially, transitioning them from the Paused-Initialized state to the Running state. The Engine, running in the background, will iterate call the input, process and output methods of the modules in the background.

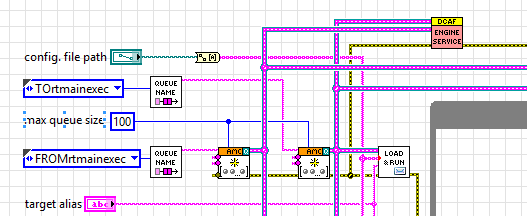
The code in the loop continuously gets the current engine state (which is almost always running) and checks for errors to stop the loop. When the main loop is stopped, all the engines are stopped and all references are closed.

Some systems may need to use an FPGA. If so, then enable the structure on the left, choose the RIO target and bitfile, and close it after the DCAF engines are shutdown.

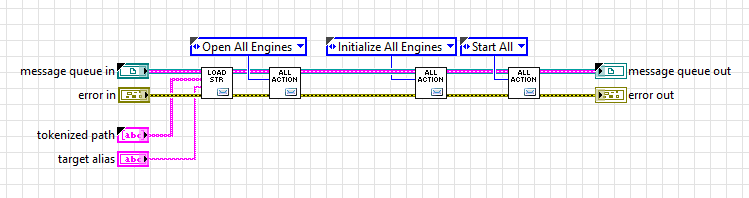
### DCAF Dynamic Execution Template

The Dynamic Execution Template is more complex than the Basic Template because it demonstrates how to launch the DCAF engine communication loop as an asynchronous process. This template creates a library for a background engine service and supported commands in addition to a *main.vi* and *includes.vi*. A unique copy of this library is created as an output of this template, and customization of this library is encouraged.

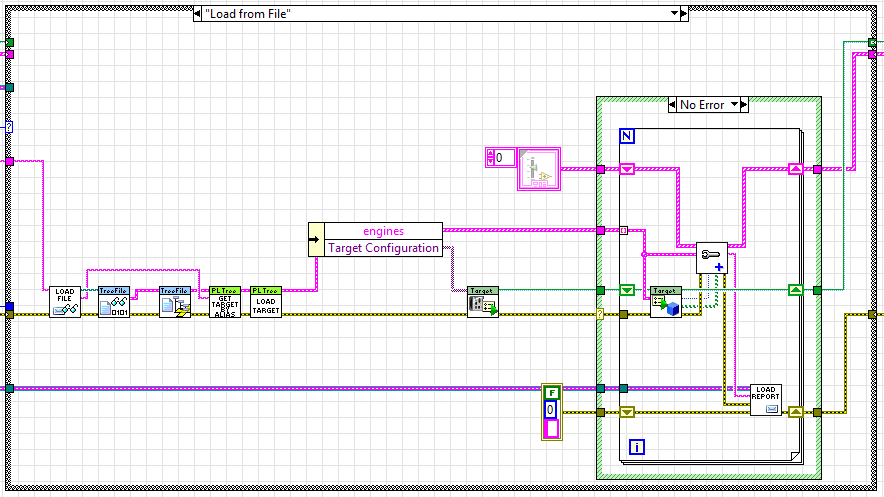
The background service that is provided uses two queues created using the AMC library. One queue is for receiving commands and the other is to provide a response.



The *load from file and run.vi* within the *main.vi* enqueues an initial set of commands to the background service: Open All Engines, Initialize all Engines, and Start All Engines.

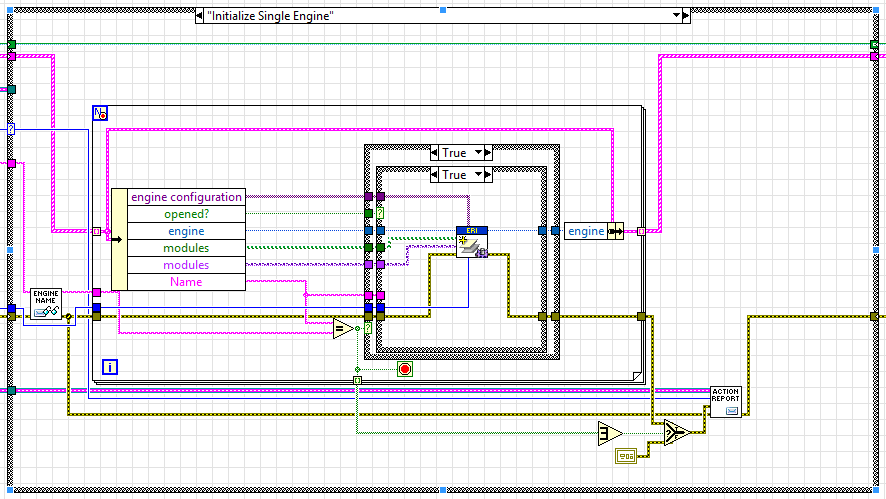


The *Engine Service.vi* uses the same API as the Basic Execution template to interact with the DCAF engine. Notice that the first vi called is *send load from file.vi* which sends the Load from File message for the engine service to process.

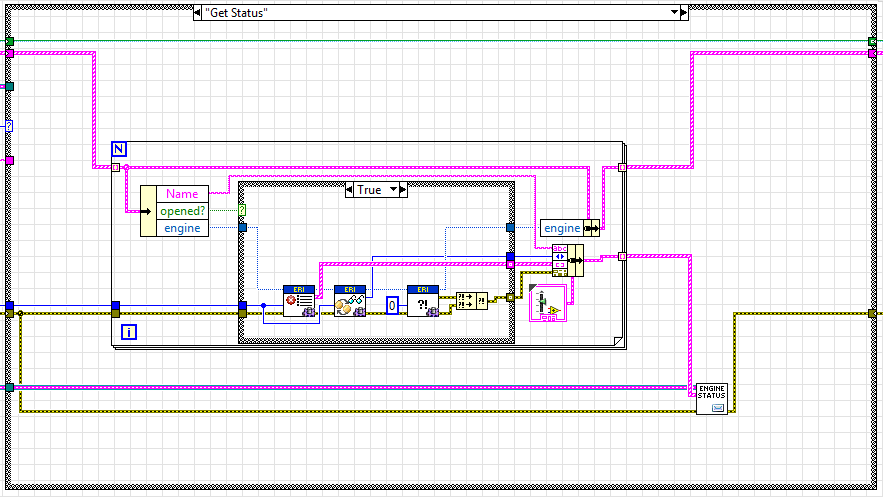


While the code here looks different than the starting sequence of the Basic Execution template, note that this is a lightly modified version of the Basic Execution Template’s *load and initialize target with engines.vi*

The next messages from the *load from file and run.vi* open the engines, initialize them and start them:



Additionally, the *Engine Service.vi* implements a Get Status method to get information about all of the current running engines – note that it uses the same functions as the Basic Execution Template:



## Creating New Modules

Most DCAF applications will require the creation of new plugin modules. Information on this process is provided in detail in the next chapter.

## Using the DCAF Configuration Editor

Use the Configuration Editor to create a Configuration File (.PCFG). The Editor will be used to add targets, engines to these targets, and modules under these engines. To use the Configuration Editor, access it from within LabVIEW by going to **Tools >> DCAF >> Launch Standard Configuration Editor**. Make sure the Configuration Editor knows where to find all desired plugin modules before configuring the application. The editor must be told where to find plugin modules on disk if they aren't saved in the default search location within vi.lib. If using custom plugin modules stored elsewhere, add their top-level folder paths using **Tools >>** **Edit Plugin Search Paths.** For a complete walkthrough of creating a configuration file from scratch, see the DCAF Hands-On.

### Adding Systems, Targets, and Engines to the configuration

The top-most node in a DCAF configuration hierarchy is the **System**. The **System** node allows a user to manage the version of their configuration and add a system description. These fields are entirely optional and should only be used if they will be kept up to date. There can only be one System node in a configuration and it is added to all new configurations by default.

**Targets** represent hardware that will execute a DCAF configuration. A new Target should be added to a configuration for every unique execution personality. Much like Main VIs, there may not need to be a one-to-one mapping between Targets in the configuration and physical execution targets. For example if four controllers can reuse the same configuration, use only one Target in the DCAF configuration to represent all four. The Target has additional properties that can be specified, all of which are optional. The IP Address is used by the Deploy Tool and may be referenced by other modules in the system, but is otherwise optional to specify. The Target also allows users to specify FPGA settings and other settings used by the Deploy Tool to transfer required dependencies. The primary purpose of a Target is simply to associate an **Engine** with where it will execute.

In most cases each **Target** should only contain one **Engine**. There are a few notable exceptions to this that include the need for modules to be triggered by different timing sources (each engine can only have one) or the desire to have a collection of functionality with its own independent state management. As a general rule, start with only one Engine for each target and add more only when needed. The default engine for DCAF is the Standard Engine.

An Engine contains all information for Tags, Mappings, timing, error handling, and more. However it is generally best to configure this information after first adding and configuring all Modules.

### Adding and Configuring Modules

Modules can vary widely in their capabilities and often require unique steps for configuration. The best way to learn more about a module and its configuration steps is to review its documentation. A list of modules and their documentation is located at [List of Available DCAF Plugins](http://forums.ni.com/t5/Distributed-Control-Automation/List-of-Available-DCAF-Plugins/gpm-p/3538587).

### Creating Tags

Despite their varying capabilities, most modules require the connection of their Channels to the Engine's Tags using Mappings. When connecting an input channel of one module to an output channel of another, only one Tag is needed. Many modules support the creation of new Tags directly from their UI, but which of the two modules, the writer or the reader, should create the one Tag needed to exchange data?

There is no single right answer to this question, but there are common approaches to take. The first approach is to have every writer to a Tag create the Tag. The main benefit of this approach is that it ensures no Tags are duplicated (because every Tag must have a single writer). One drawback is that multiple modules must create Tags and that each module only creates Tags for its Input and Result channels. Not every module may support this capability. Another drawback is that Tag names are usually copies of a module's channel names such that the list of Tags that get created have varying naming schemes.

An alternative approach is to have either the processing modules or the I/O modules create Tags. This will result in fewer modules creating Tags and more consistent naming, but may still lead to missing or unnecessary Tags.

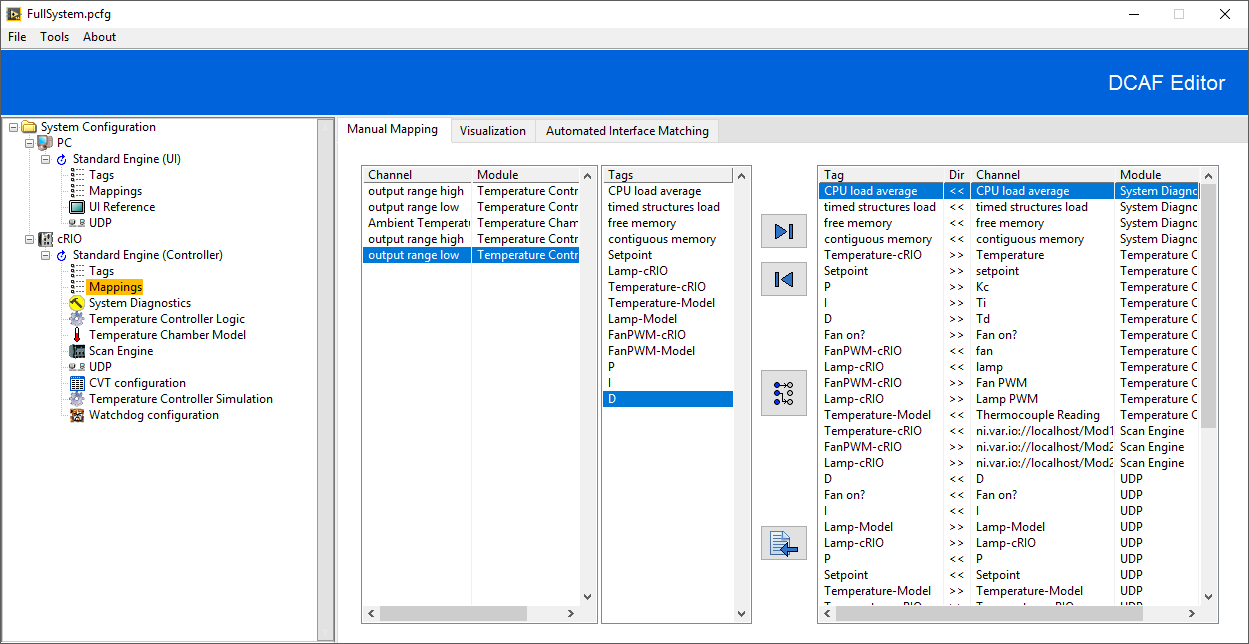
Regardless of the approach taken, some Tags may still need to be added, removed, or renamed manually from the Tag node under the engine.

### Complete and Verify Engine Mappings

Tags created automatically within a Module's UI generally include a mapping connecting that module's channel to the created Tag. However mappings must still be made for the other end of that Tag's connection. Once again, these mappings can often be made directly from a module's configuration UI. This is generally the best path to take when supported.

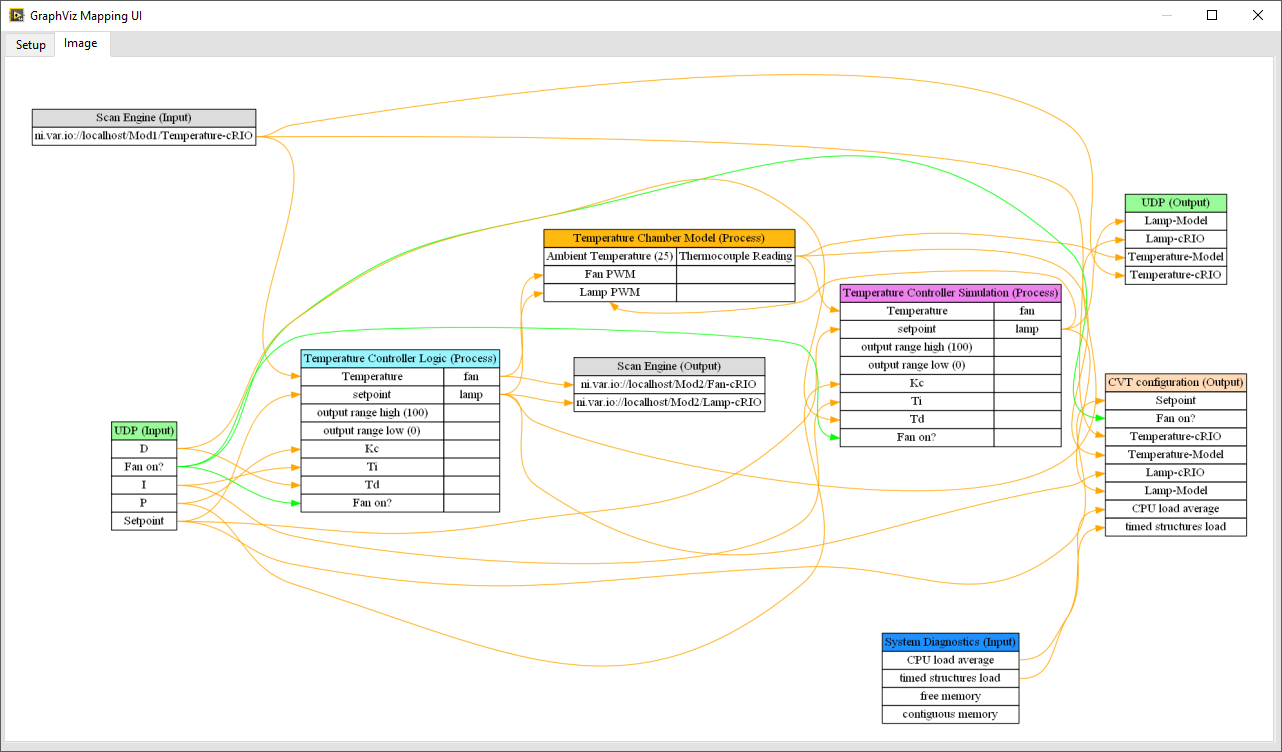
Mapping can also be generated manually using the Mapping UI under the engine.

Check for unmapped channels in the mapping UI. Use the GraphViz tool to double-check connections. GraphViz is an open source tool for visualization. DCAF uses it to show you how you have mapped your tags. To use the tool, press the visualization button. If you don’t have GraphViz installed and added to your windows PATH environment variable, you will need to manually do it.



The GraphViz button is highlighted in red.

Hitting the button produces a visualization like the one below. The tables represent the modules running on a system. The rows in the tables are the channels for those modules. The arrows are the tags, with their color indicating their data type.



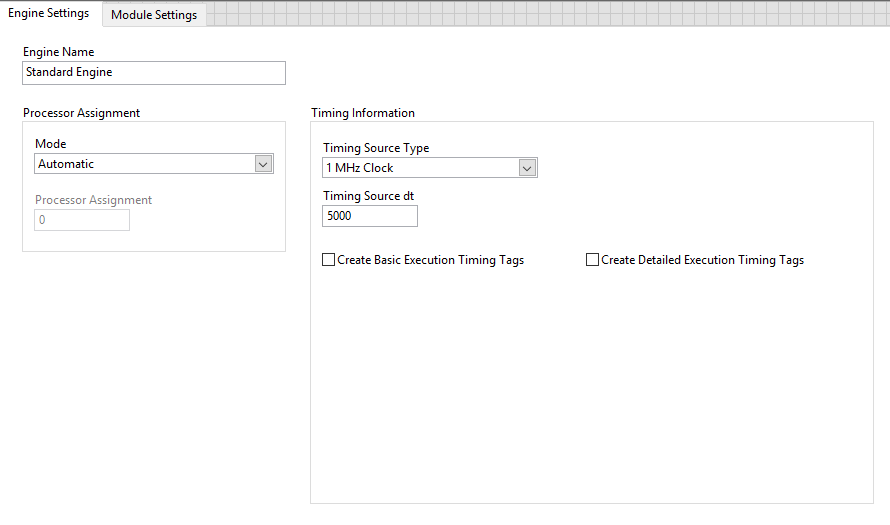
The visualization produced for the cRIO target of the FullSystem.pcfg example configuration file that ships with DCAF Core.

### Specify Engine Execution Configuration

The standard engine has some required configuration for DCAF applications. For detailed documentation, [see this page](http://forums.ni.com/t5/Distributed-Control-Automation/Standard-Engine-Documentation/gpm-p/3539201). To get started, pick a timing source appropriate for your application. By default the 1 MHz with dt of 5000 means that the engine runs 1 time for every 5000 ticks. This is equivalent to one execution every 5 ms.

If required, you can also specify which CPU processor core the engine will run on. Usually, LabVIEW and the operating system will choose an optimal behavior and automatic is appropriate.

The create timing tags option swill create tags that may be useful for ensuring your application performs deterministically enough for your application.



### Error handling options/best practices

One of the tasks of the DCAF engine is to handle errors thrown by any of the modules that it executes. The framework divides errors into 4 categories: unknown, trivial, recoverable, or critical. Each module is expected to categorize its own errors for the engine which will then handle them.

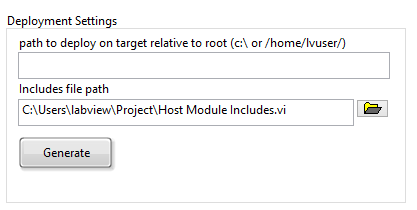
Each engine can be configured for a specific action when an error of a specific category occurs within a specific module. Each type of engine can define its own list of error actions. Configuring error actions carefully is an important part of any DCAF application. See the [Standard Engine Documentation](http://forums.ni.com/t5/Distributed-Control-Automation/Standard-Engine-Documentation/gpm-p/3539201) to learn more about its error handling capabilities.

## Loading Dependencies and the Module Includes VI

DCAF is a plugin-based framework without static dependencies between the framework and its modules. Unlike the default behavior with subVIs in typical LabVIEW programming, LabVIEW won't automatically pull in the source code for each plugin. Instead it is up to the developer to ensure proper source code dependencies are included. Dependencies must be loaded into memory before the DCAF configuration file is parsed. There are a few ways that this can be done.

Recall from earlier in this chapter that both DCAF execution templates include a VI called *Module Includes*. This VI is intended to house the runtime class object of each DCAF module included in that Target's configuration. The static call to this VI within the Main VI is enough for LabVIEW to automatically pull in all source dependencies needed to run the application. This approach automatically loads those dependencies into memory and includes the dependencies within an EXE build.

The *Module Includes* VI does not have to be modified manually. Instead, after creating a Main VI from either of the execution templates and completing the application's configuration, the Configuration Editor can be used to script this VI. This can be done from the editor UI for the target itself under Deployment settings; first choose the filepath of the *includes.vi* and then click on generate.



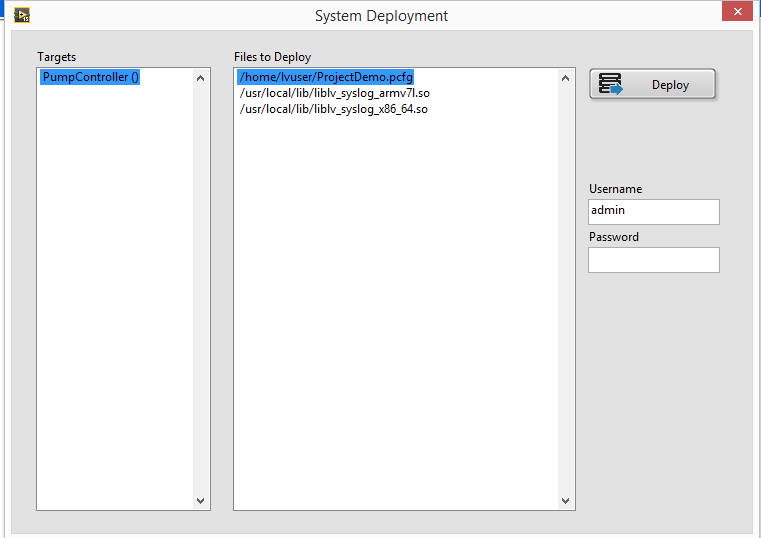
The *Module Includes* must be kept up to date with any dependency changes in the configuration. If a new module is added, or an existing module is replaced with a different one, the *Module Includes* will need to be remade to reflect the changes.

The *Module Includes* approach to loading dependencies is simple, reliable, and recommended whenever it can be used. The main drawback to this approach is that plugins and their source must be known up front when building the DCAF executable. A change to a plugin requires building a new EXE. It is possible to instead build each plugin independently and load them dynamically. This approach has the benefit of allowing plugins to be updated and added without modifying the EXE. In fact, the EXE could be designed to load new plugins and a new configuration without having to restart. While these use cases are possible with DCAF, DCAF currently does not provide any tools to simplify this complex process.

## Deploying a Configuration File

Running DCAF engines on your development PC is straightforward, but running on remote targets such as a Compact RIO require additional steps to stage all the files correctly. Before running the code on a remote target, the *.pcfg* configuration file needs to be deployed. This can be done in two main ways – either using the DCAF built-in tool in the editor, or invoking the file transfer tools yourself.

The DCAF editor can deploy saved configuration files using the Deploy Tool found from **Tools>>Deploy Tool**. If there are unsaved changes, the editor will prompt you to save. Before using this tool, ensure that **SSH is enabled** on that target's settings in MAX as this is required to create a connection. Once open, the Deploy Tool will display the following **System Deployment** window.



This tool will populate the files that must be deployed for the configuration and have a place to enter the Username and Password to access the target. Note that the targets on the list are selected by name – the connection uses the IP set within the target’s configuration window in the Editor:



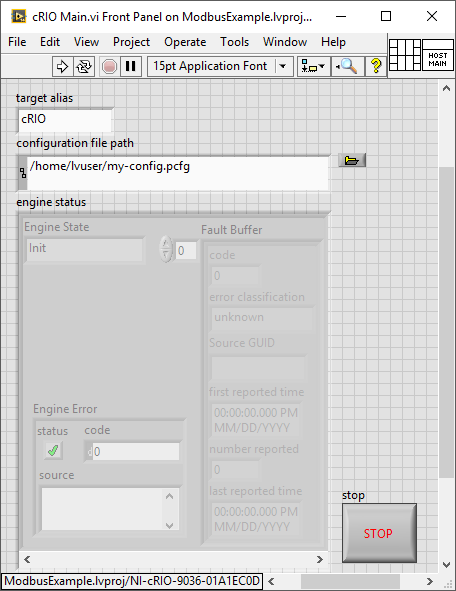
See the section titled 'Deployment Issues' if there are errors when deploying your files to your target.

It is also possible to manually push the necessary files using WebDAV or SFTP. To move files using WebDAV, follow the [Using WebDAV to Transfer Files to Real-Time Target](http://digital.ni.com/public.nsf/allkb/4EBE45E8A816B19386257B6C0071D025) article and put the files in the correct location as per your application.

Alternatively, it is also possible to write code for the target to, on startup, browse to a specific network location or check a shared drive and pull a new file if there is a newer version. This could be done using a network share, or using the WebDAV APIs within the target to pull new files from a location if necessary.

## Running the Main VI and Debugging from the LabVIEW IDE

Once you have created a main VI, built new plugin modules if necessary, created a configuration file, scripted the includes VI, and deployed the configuration file, the application should be ready to run. To run the main VI, select the file path for where the .pcfg file is saved on the execution target and the target alias of that target as specified in the Configuration Editor. If these values are constant, consider making them the default value of the controls or changing those controls to diagram constants.



Main VI for DCAF Application. The Target Alias is set to match what is in the configuration file. The path is the path on the target to the configuration file.

If the run arrow is broken on the main VI, LabVIEW will prompt you with a list of errors it found. Resolve each error from the top down and your VI should be runnable.

### Deployment Issues

You may also run into problems when attempting to send files to the target via tools such as the Deploy tool within the editor. If so, then check that SSH is enabled in the target from MAX.

During deployment, it is possible see problems with classes being loaded with errors, found and not loaded, or not found.

If classes are loaded with errors, there may be multiple versions of a specific class in the target – to deal with this clear the target’s object cache from **Tools>Advanced>Clear Compiled Object Cache**.

If classes and dependencies are found but not resolved, drivers may be missing from the target.

If the classes or dependencies are not found, it may be a myriad of different issues. First Make sure that the includes VI is up to date with the current config by rerunning the includes vi script.

If the deployment completes with errors, it could be possible that the connection is lost due to a target’s crash. You can check whether the process has crashed by using the command **ps | grep “lvrt”**.

### Functionality Debugging

Should a DCAF system build and deploy correctly but not work as expected, there are many techniques for correcting the behavior. This section details some of the techniques.

Troubleshooting is often made easier by reducing the amount of code in which to find the problem. DCAF enables this by allowing an application to come together one module at a time instead of all at once. Piecing together your system like this can ensure that each module that is added is acting as expected before putting everything together. If system behavior is not as expected, the problem can be usually be isolated to the recent changes.

Because DCAF is heavily configuration-based, it is important to first validate the configuration before troubleshooting source code. One of the most common mistakes using DCAF is incorrectly configuring tags and mappings. To verify mappings, visualize them with the built in GraphViz tools described earlier in this chapter. The more advanced runtime example lets you modify tag mapping and reload the configuration file without stopping, so it consider using it for iterative debugging.

Only after being confident in the validity of the configuration should source code debugging be considered. Most source code in DCAF is set to reentrant execution with debugging turned off for optimal execution performance. Unfortunately these settings can make debugging tricky, especially in LabVIEW Real-Time. However, it is possible to work around this. The easiest way to do this is to open the module runtime VI you are interested in inspecting. Then open the File >> VI Properties page, switch to Category >> Execution. Check ‘Allow Debugging’ and change the Reentrancy to Non-reentrant execution. This will enable standard LabVIEW debugging tools for that VI while it is executing. When done debugging, make sure these settings are reverted to their original values for optimal performance.

Another method for troubleshooting is to use Syslog, the standard logging solution for event recording on Unix-like systems. More information on Syslog [can be found here.](http://www.ni.com/example/30980/en/) DCAF makes extensive use of Syslog to enable debugging by logging events in the standard engine. It uses a simple wrapper VI that you can include in your own custom plugins. More information on using the library can be found on the article [DCAF Event Logging with Syslog](http://forums.ni.com/t5/Distributed-Control-Automation/DCAF-Event-Logging-with-Syslog/gpm-p/3586880). To get started quickly, you can find a convenient logging VI in < vi.lib>\NI\Syslog Wrapper – Linux\write syslog.vi.

You can also export your tag values in a few other ways to verify they are correct. If you are already using a Current Value Table in your application, consider routing some of your tags to it to see their values. You can also use the TDMS Logging Module to log values of your tag bus to inspect for debugging.

## Building and Debugging an EXE

For running your application headlessly, you should build it into an executable. Please view the [CompactRIO Developer’s Guide](http://www.ni.com/compactriodevguide/), section [5](http://www.ni.com/compactriodevguide/sec5.htm), for more information on Real Time Systems and deployment.

For projects requiring an EXE, be sure to build the EXE early and regularly. Build issues are difficult to debug. By building regularly (for example, with each new module added to your application), you can identify what additions or changes caused your application to break. If your application is already complex with many modules, it will be difficult to determine which broke your application.

If a build issue is encountered, start by attempting to reproduce the issue with the smallest amount of code possible. If you haven’t been building regularly, it may be necessary to use a binary search to figure out which section of code is causing the problem. This can be done by using the ‘Diagram Disable’ structure to remove sections of the source code until the problem section is identified. Additionally, look for any circular dependencies in the source code and try to remove them if possible. Once the problem section of code is identified, it may be necessary to contact National Instruments support for further troubleshooting assistance.

Once built successfully, debugging an EXE is similar to debugging your application run normally except that debugging with the LabVIEW IDE tools can be much harder. Instead, consider using Syslog, TDMS, and other tools described in section 3.6 that don’t rely on the LabVIEW IDE. Issues are also commonly caused by changes to relative VI paths that occur during the build, and UI references used in property and invoke nodes that are no longer valid.