# Component Overview

This Chapter will provide a big picture overview of DCAF and its many components. It begins by describing the framework's nomenclature. Next, it explains the high-level architecture and use-cases for DCAF. After that will be an overview of each of the major framework components with links to additional documentation for understanding them in greater detail.

## Nomenclature

**Target:** Computing hardware that executes a software application. The term 'Target' is used in the LabVIEW project to identify where a LabVIEW VI will execute and within the DCAF ***Configuration Editor*** to contain one or more ***Engines***.

**Engine:** State machine that executes ***Modules*** as defined in the ***Configuration Editor*** and provides a namespace for ***Tags****.* The Engine also passes data between its ***Modules*** according to its ***Mappings***.

**Module:** A collection of code of varying functionality that executes within an ***Engine*** and that can interact with the ***Tags*** of that ***Engine***. Some standard Modules are installed with DCAF, but users can also create their own.

**Tags:** Engine-scoped latest-value variables that can be accessed by any ***Module*** within that ***Engine***. ATag can be defined as a connecting point between ***Channels*** from different ***Modules***. Each Tag must have a specified datatype and be given a name that is unique for its scope.

**Tag Bus:** The collection of tags that the engine passes around to each of its modules.

**Channels:** A ***Module's*** input and output data which can be connected to Tags through ***Mappings***. Channels are like ***Tags*** in that they are latest-value variables, except they are scoped to a specific ***Module*** and have a category based on how their data is exchanged with the engine (input, processing parameter, processing result, output).

**Mappings:** Mappings are the connections between ***Tags*** and ***Channels***. If you want a specific ***Channel*** to write or read a value on a specific ***Tag*** you will have to map them.

**Configuration Editor:** A LabVIEW program used to visually define the functionality of each ***Engine*** and ***Module*** in DCAF.

## Architecture

A DCAF application is composed of four main pieces: a system configuration file created by the Configuration Editor, a Main VI to read and execute the file as well as handle any non-framework tasks, one or more DCAF Engines, and one or more plug-in Modules executing within those engines.

DCAF Execution Engine

Run State

Main.VI

Configuration Editor

Configuration API



File API

Engine Command API

Other Processes

Process

Inputs

Plugin1

Plugin2

Outputs

Plugin1

Plugin2

Plugin2

Plugin1

Tag Bus

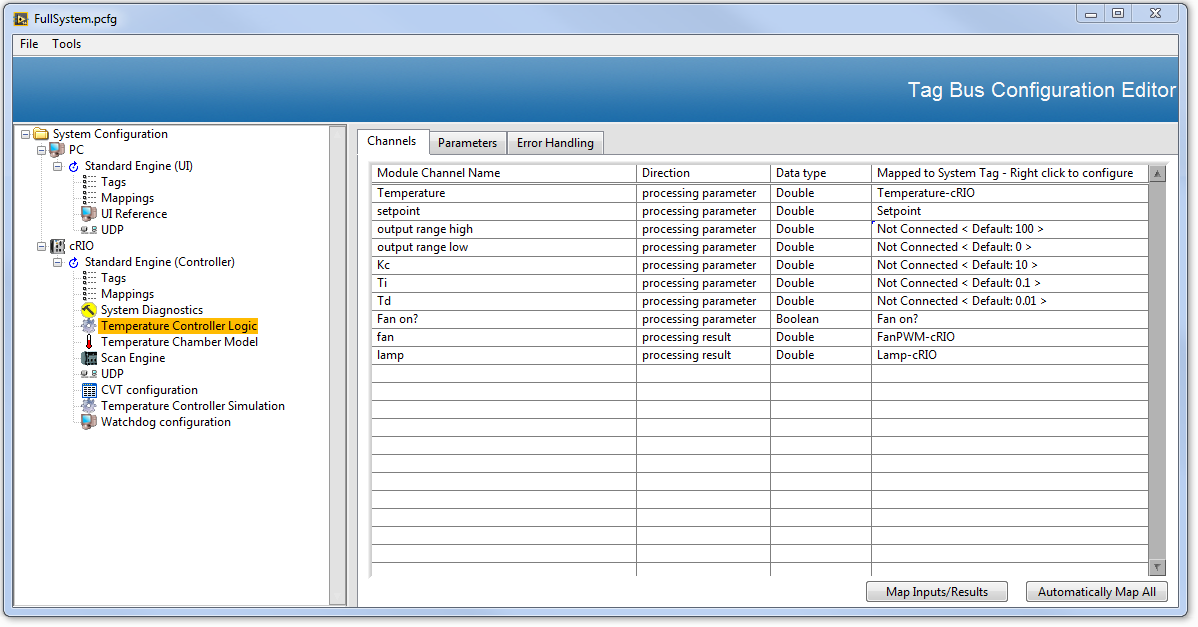
Idle

Configuring

Safe State

Shutdown

API Calls



PluginUI

Error Handling

In the architecture diagram, all code shown in **Blue** is provided out of the box by DCAF. This code can generally be used as-is and is not intended to be modified. Code shown in **Yellow** are the plugin Modules used to customize the behavior of an application. Some modules are provided with the framework, but most applications will require the creation of additional modules. Code in **Green** runs parallel to the framework and represents customization of the Main VI. Any application functionality that does not fit into an Engine's execution model can still be developed outside of the framework. Common examples include data streaming and waveform acquisitions.

Out of the box, DCAF provides some commonly used functionality with the modules available online. These modules allow for interfacing with Scan Engine, common industrial protocols, and other functionality. A key step when building a DCAF application is to design and implement any additional plugin modules needed that are not already built for you. For example, you may use an industrial communication protocol that has not already had a module built for it. Custom control algorithms also need to be implemented as modules. The module design process is described in more detail later in the guide. Once all plugin modules are developed, the behavior of those modules and their exchange of data is defined using the Configuration Editor.

The output of the configuration editor is an XML-like file that can be transferred to one or more targets. This file contains all of the information needed for initializing an engine and its plugin modules to achieve the configured system behavior.

The Engine Command API is used from the Main VI to pass the configuration data from the file to one or more engines. The Engine Command API is then used to walk the engines through their state transitions and monitor their status.

In the run state, each Engine will sequentially call the input method for each plugin that implemented it, followed by the process methods, and then the outputs. Sequential execution ensures that no method executes before its inputs are available. It also allows results from one processing step to arrive as parameters to the next processing step without incurring a cycle delay. Each engine also handles errors for each of the modules that it calls. These error handling options are very flexible and are explained in more detail later in the guide.

All data communication between modules is handled by the engine. In other words, the engine moves data between plugin modules, and those modules can't access each other's data without the help of the engine. This constraint ensures predictable and intentional behavior within the engine by eliminating race conditions and data coherency issues that would otherwise be possible.

This combination of out of the box functionality, flexible configuration, and correct-by-construction constraints lead to high quality and feature-rich applications completed in much less time.

## Appropriate Use-Cases for DCAF

DCAF is a single-point I/O architecture designed for control and automation applications. It natively handles data in the form of tags which can be single values or arrays, but not buffered waveforms. The framework does not have built-in mechanisms to handle buffered data or to route messages. Therefore, the DCAF engine is not suited for waveform acquisitions or command/response systems. Buffered functionality can still be implemented outside of DCAF, however. DCAF also does not have built-in test or modeling features required by most Real-Time test applications.

DCAF provides tools to simplify the job of programming a full application, but it is not turn-key software. DCAF does not replace the need for LabVIEW and LabVIEW Real-Time programming skills. DCAF also does not eliminate the need for programming with LabVIEW FPGA.

## Component Overview

### Modules

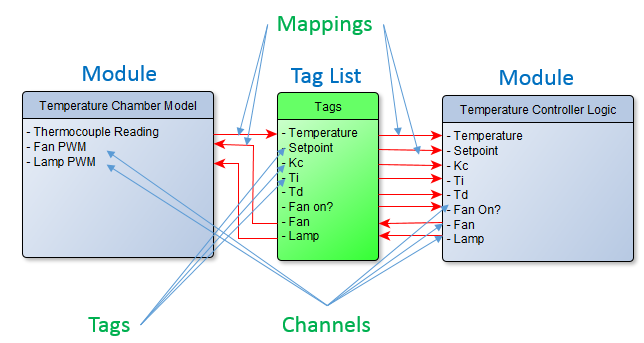
Modules are the pieces of a DCAF system that interact with the data flowing through the engine. Modules can act as I/O, pulling data into or pushing it out of the engine, or it can act upon the data for control algorithms or other applications. Some Modules are installed with DCAF, others can be found in the DCAF community, and others can be custom built for a specific application. There are two main designs of modules: dynamic modules and static modules. Dynamic modules are designed to have arbitrary numbers of tags connected to them, but static modules have a fixed number of tags connected. Dynamic modules are therefore useful for I/O, where different applications might use different channels. Static modules are useful for control algorithms such as a PID algorithm, where the numbers of inputs and outputs will be the same between applications. If a module doesn't send receive or send information out of DCAF, it is also called a Processing Module.

Some examples of dynamic modules built for I/O are FPGA, Scan Engine, Modbus, EtherNET/IP, TCP, UDP, or Current Value Table (CVT). Some dynamic modules for logging or communication are the TDMS or Web Services modules. Available static modules include Scaling and Alarming modules. For a full list of modules and documentation please [see this page.](http://forums.ni.com/t5/Distributed-Control-Automation/List-of-Available-DCAF-Plugins/gpm-p/3538587)

Modules contain Input, Process, and Output methods that are sequentially called by the engine in the main loop, as well as Init and Close methods that are called once – how this happens is discussed further in the engine section.

Modules do not directly modify the data held within the engine's tags but instead read and write to a set of Channels. These Channels are very similar to tags, but the main difference is that they have a direction – for example, data that are acquired or generated in the module and sent to the engine are **Input Channels**, while those that are sent from the engine to the module are **Output Channels**. Similarly, **Processing Parameters** and **Processing Results** are the inputs and outputs of a Processing Module respectively. The engine then takes care of taking the value of a Channel and passing it to an engine Tag (in the case of an Input or Processing Result), or vice-versa (in the case of an output or Processing Parameter). The link between the Channel and the Tag is called a **Mapping**, and it is defined using the **Configuration Editor**.

Take the following example of a temperature control system to clarify the previous terminology. The ***Temperature Chamber Model*** is the module that contains the I/O for the system. The ***Temperature Controller Logic*** module is the control algorithm that computes the value of the outputs based on the inputs. ***Temperature Chamber Model*** has an Input Channel called ***Thermocouple Reading***– the module implements reading from a thermocouple and puts the value into the **Thermocouple Reading** channel. This ***Thermocouple Reading Channel*** is mapped to a Tag called ***Temperature*** – the engine will then take the value that the module places onto the channel and put it on the tag. Then the ***Temperature*** Tag’s value is passed by the engine to ***Temperature***, a Processing Parameter Channel that belongs ***Temperature Controller Logic***. This processing module has an algorithm inside of it to determine the values of the **Fan** and **Lamp** Processing Results Channels, which are mapped to the **Fan** and **Lamp** tags. These tags are then mapped to the Output Channels **Fan PWM** and **Lamp PWM** of the **Temperature Chamber Model.**



This example also helps demonstrate the value of DCAF for making modular code. While the ***Temperature Controller Logic*** module might be ideal for one project, the next project building a similar system may require a different algorithm. DCAF's modular design allows swapping out that module for a different one, allowing reuse of the ***Temperature Chamber Model*** for future projects.

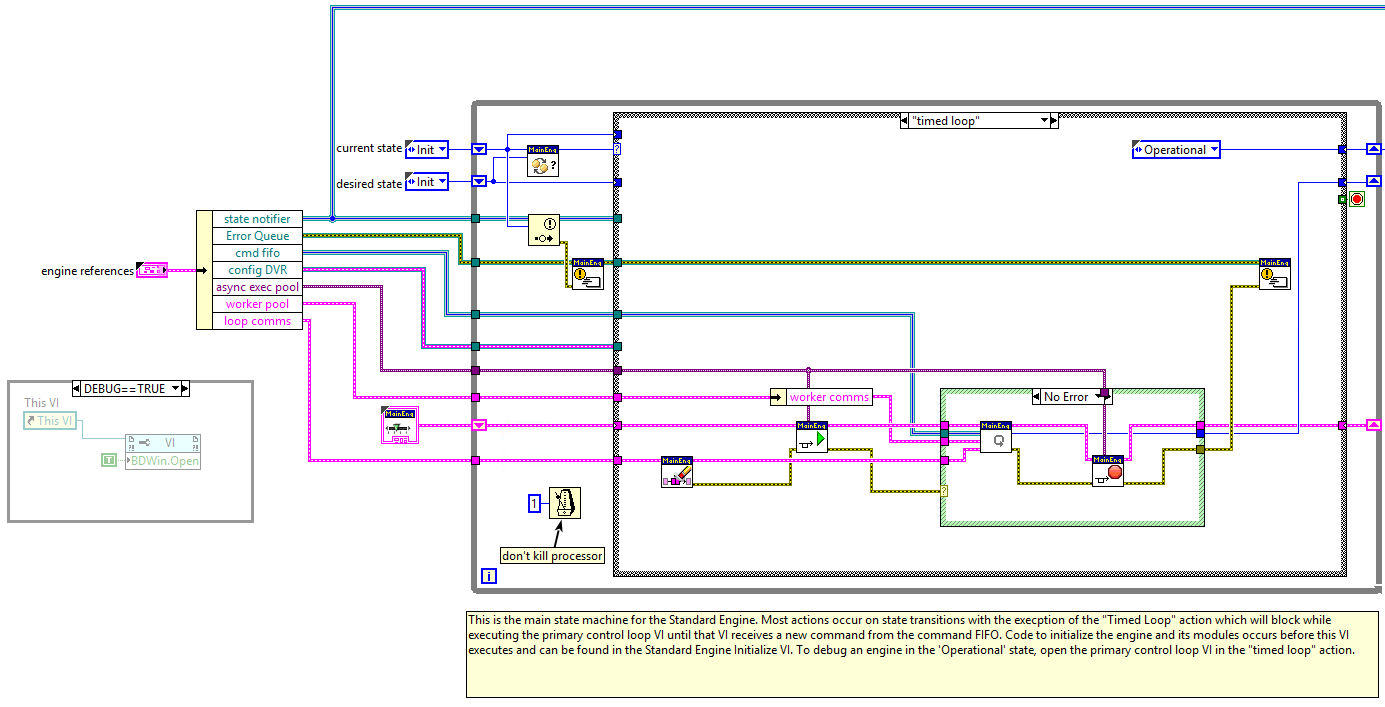
DCAF allows modules to be built quickly because of a series of templates and scripting utilities. The process of module creation and their associated templates will be covered in-depth later in this guide.

### Engine

Each Engine is a background process that contains a state machine. It is responsible for calling the modules Input, Process and Output methods as well as other activities such as fault recovery. The engine’s behavior is defined in the editor by the user and it is launched and communicated to using API calls.

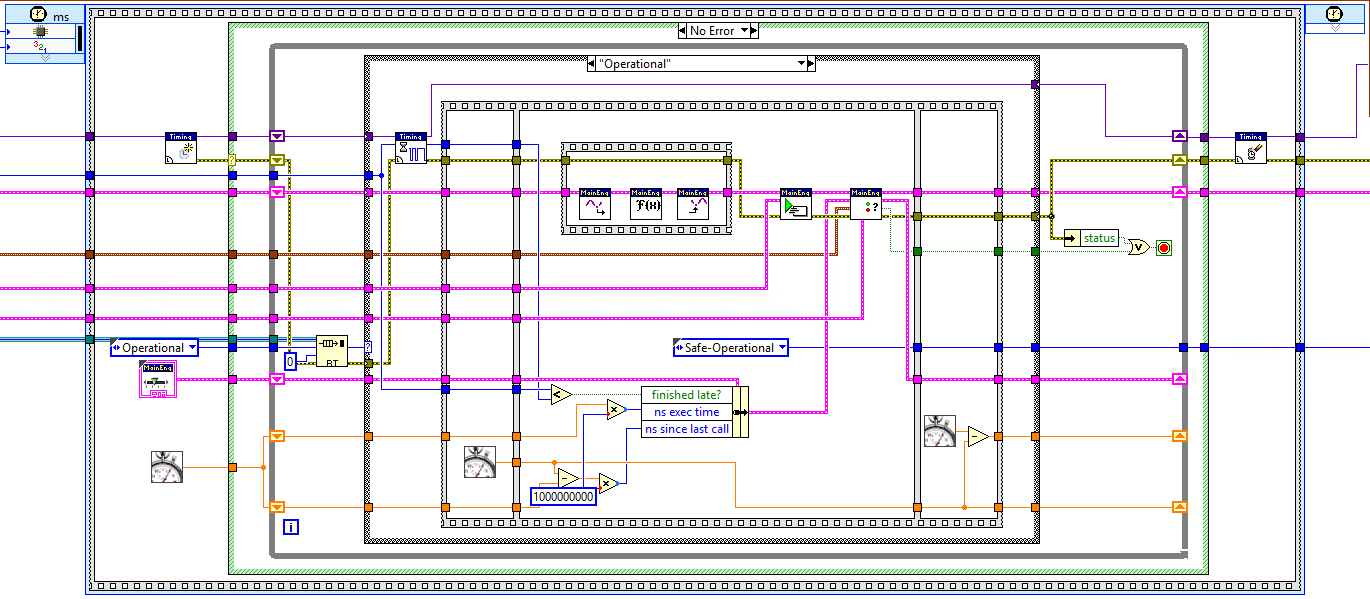
DCAF is built upon a plugin scheme – this means that there is an abstract Engine Runtime Interface that allows you to override the Standard Engine and provide a different implementation if necessary. There is currently only the Standard Engine implementation, however.

The figure shows the code that is being executed within the engine:



Main engine.lvlib:Standard Engine.vi

The run state is the case called “timed loop”. It contains a **timed structure** which, in turn, contains the input, process and output operations of each individual module. Note that it is guaranteed that all input methods are called before the process methods and that those are called before the output methods:



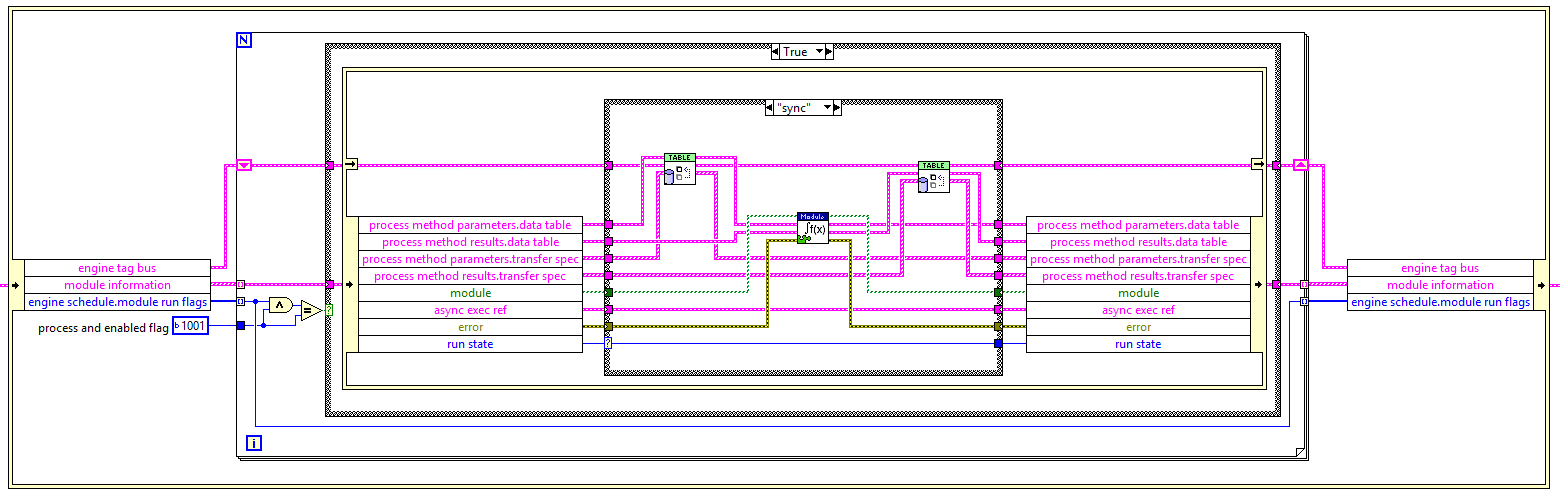
main engine.lvlib:primary control loop.vi

The engine follows this sequence of steps:

1. Check for a message to stop.
2. Wait for next trigger of the configured timing source to fire.
3. Read all module input channels and store the gathered data on the tag bus (write to Tags).
4. For each module with a processing method, read needed processing parameters from the tag bus, call the process method, and write the processing results back to the tag bus.
5. Get the final data from the bus (read from Tags) and call the Ouput method for each module.
6. Choose which plugins execute on the next iteration. (It is possible to configure some modules to run less frequently.)
7. Check for errors and handle them.

Note that it is a timed structure and not a timed loop because DCAF allows you to choose the timing source – the engine waits on that clock with the *wait on clock.vi* with the *usec timing source.lvclass, scan engine timing source.lvclass* and *ms timing source.lvclass* already having been built into the system. For more information on the timing source, please view the Timing Source section.

The wire connecting the input, process and output is the cornerstone of DCAF and contains all the tag data. It is the global repository within this engine for all the data – each module registers for a smaller subset of the available data. Looking at the process operation’s VI:



The first VI in the case structure, *transfer table data.vi* takes the full data from the engine and pulls out the individually registered data and passes it to the module. Note that since it’s a processing module it produces data which is then merged with the engine’s tag data using the same *transfer table data.vi*. This transfer is **by value**, not a pointer, and takes advantage of LabVIEW’s dataflow optimizations and allows for having multiple engines each with its own, scoped data.

This scheme of sequential inline execution of plugin modules does not preclude an engine from executing a module asynchronously. The Standard Engine can generate a separate free-running thread in which it can call and collect module data without blocking the main engine loop. This asynchronous setting is configurable from the engine's settings.

### Timing Source

Each instantiated engine has a timing source chosen within the configuration editor. This source is used to time the loop’s execution. The Engine uses the Loop timing Source API which defines properties and methods for triggering loop execution. Ultimately, this is a similar experience to using LabVIEW Timed Loops in RT.

There are 3 main timing sources to choose from: The 1 MHz clock, the 1 kHz clock, and synchronizing to scan engine. The first two options have a concept of time since they are based on the OS clock, but synchronizing to scan engine gives a time relative to the RSI clock in the FPGA.

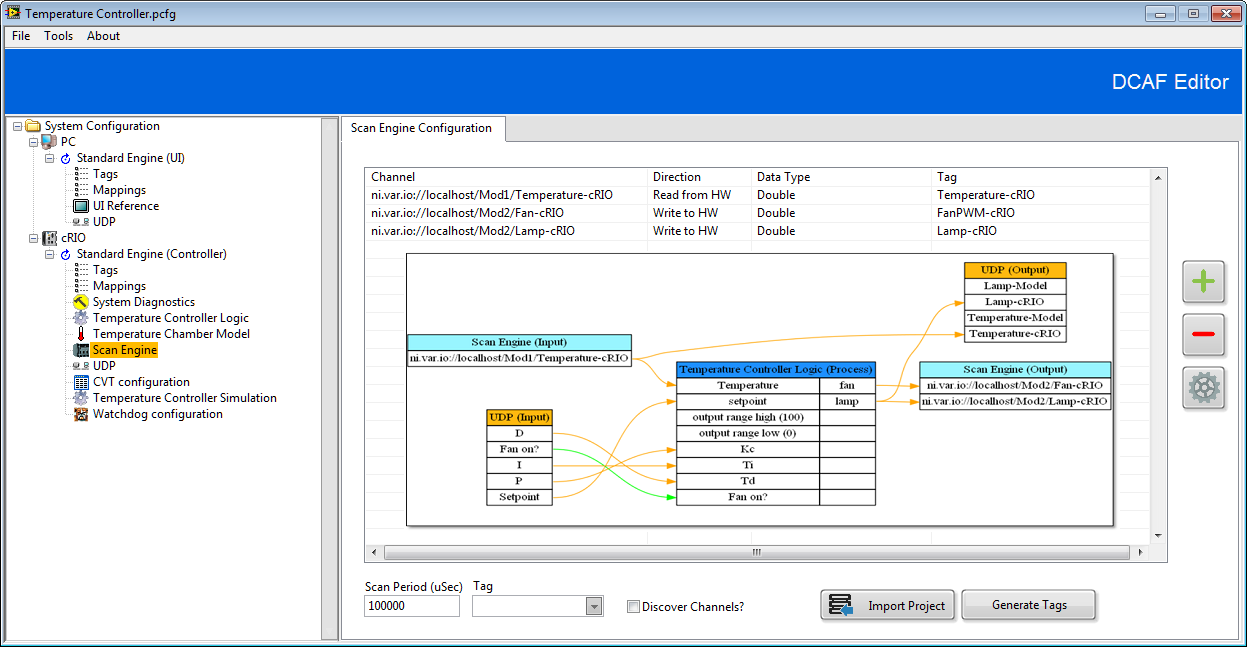
It is possible as well to build your own timing sources by overriding the timing source API’s methods.

### Configuration Editor

The configuration editor is used to configure the multiple targets that are part of a DCAF system, to add engines to each target and to add modules within that engine. The editor maintains a list of the tags in the system and how they map to the channels in the different modules. It also provides features such as automapping and displaying a summary of the system configuration.

The DCAF Editor shows a hierarchical view of a DCAF system on the left tree and information on the selected item on the right window; these configurations can be saved and loaded through the **File** menu. The different items in the left pane's tree are called nodes.

The editor can also show the user the node data that it will save into the XML configuration through a right-click menu, as well as attempt a repair of the node in case there are errors with your configuration.



Each module has its own configuration window, populated by the module itself. These windows can be either completely automatically generated, as in the case of Static Modules, or they need to be modified and UI code must be written as is the case with Dynamic Modules. To learn more about the process of creating these UIs, please view the section on Dynamic Modules as well as the Editor Template section.

The Engine and target itself also have configuration windows. The engine has its name, assignment to a processor, as well as Timing Source, while the target has the type (Linux RT ARM, Linux RT x64, Windows, etc), FPGA settings if available, and deployment settings.

Having configured the system, the configuration itself must be saved, exported as a *.pcfg* file and deployed to every target system. At runtime, the file is used by an application to instantiate and configure all the modules before beginning execution.

## DCAF Packages

DCAF has multiple layers of packages which may be useful on their own. A [dependency graph for DCAF](http://forums.ni.com/t5/Distributed-Control-Automation/DCAF-Package-Dependency-Graph/gpm-p/3539199) can be found in the community. The DCAF Core package collects the essential packages for building a DCAF application, running examples and getting started with templates.

These packages contain APIs such as the Tag Bus API, the Loop Timing Source API, or the Engine Execution Interface. The APIs are explained in the article [Unders](http://forums.ni.com/t5/Distributed-Control-Automation/Understanding-the-Different-Components-of-DCAF/gpm-p/3537534)tanding the Different Components of DCAF article in the community; the main API used to handle data within DCAF is the [LabVIEW Tag Bus Library](http://forums.ni.com/t5/Distributed-Control-Automation/LabVIEW-Tag-Bus-Library/gpm-p/3581721).

Separate from the DCAF Core package are modules for download that implement access to more I/O and other useful features.