# Application Design

A DCAF application can be thought to consist of three types of code: pre-existing DCAF plugin modules and engines, newly created DCAF plugins specifically for your application, and custom software written outside of DCAF. When designing a new application, make a plan for which of these categories each piece of application functionality will fall into.

The best place to start in the process is to see what DCAF can cover out of the box.

## Using existing DCAF plugins

DCAF plugin modules exist to gather local I/O or remote I/O, process and scale that data, and more. See [List of Available DCAF Plugins](http://forums.ni.com/t5/Distributed-Control-Automation/List-of-Available-DCAF-Plugins/gpm-p/3538587) for a list of plugin modules found on the NI Tools Network.

For each plugin module that will be reused, it is good to try each of them in isolation to learn more about their features and uncover any gaps that may not fully meet a use-case. For example, if data is to be logged to disk using the TDMS plugin, build a small test program to test that plugin and its features. Issues can be posted to the DCAF Community or GitHub page for any features that are missing. Because DCAF and its plugin modules are open source, the features can be added directly to the existing module source code. Improvements you make to modules can be shared with all of DCAF's users thanks to its open source basis. See the section in the guide on community collaboration for more details.

Ideally a large subset of application requirements can be met by DCAF out of the box. If not, this may be a sign that DCAF is not a good fit for the project. For all remaining functionality, a decision must be made on whether the functionality should be handled by newly developed DCAF modules or built external to DCAF.

## Deciding when to build new plugin modules

A good way to get a sense of what makes a good DCAF module is to get to know the ones that already exist. Again, modules should operate on latest value data by generating new values, modifying existing values, or routing existing values. Functionality requiring any form of buffer is a sign that it may not be a good fit. However in some cases functionality that needs to read from a buffer and generate tag values (like alarming) may still be appropriate.

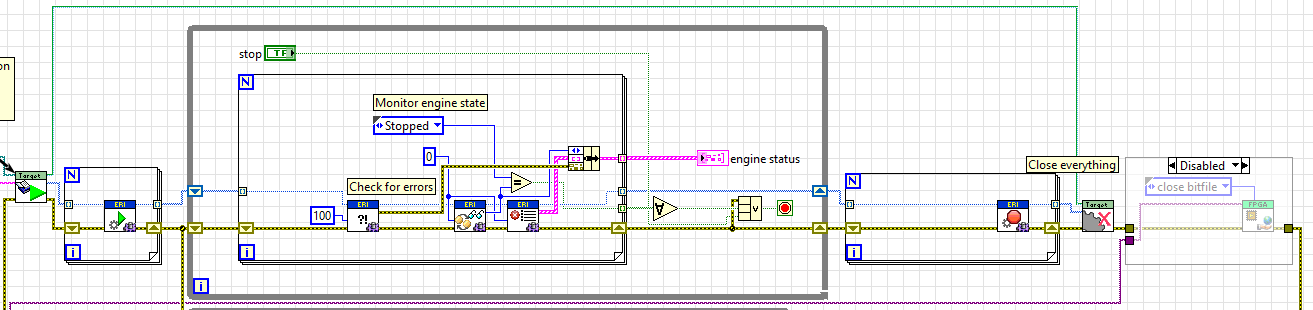
In some cases, the functionality of a module may make it reusable in other projects. This may include a module for a new type of I/O, communication protocol, or common form of data manipulation. Designing your modules to be reusable can save time and effort in future projects and should be a goal whenever possible.

In other cases, application-specific functionality may need to go into a module. Not every DCAF module needs to be highly reusable. DCAF was designed to support one or more application-specific plugins that benefit from the features of the engine that executes them. The main use-case for this is custom algorithms or state logic.

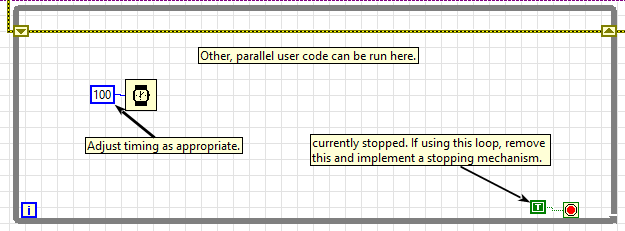
More information on designing and building new modules is provided in a later chapter.

## Adding functionality outside DCAF

If functionality is not provided out of the box, and isn't a good fit for a new DCAF module, it can still be added to the application's Main VI. All DCAF applications must contain code to gather the configuration data and start up the engines to execute that data. An example of this explained in the next chapter is the 'DCAF Basic Execution Template' pictured below.



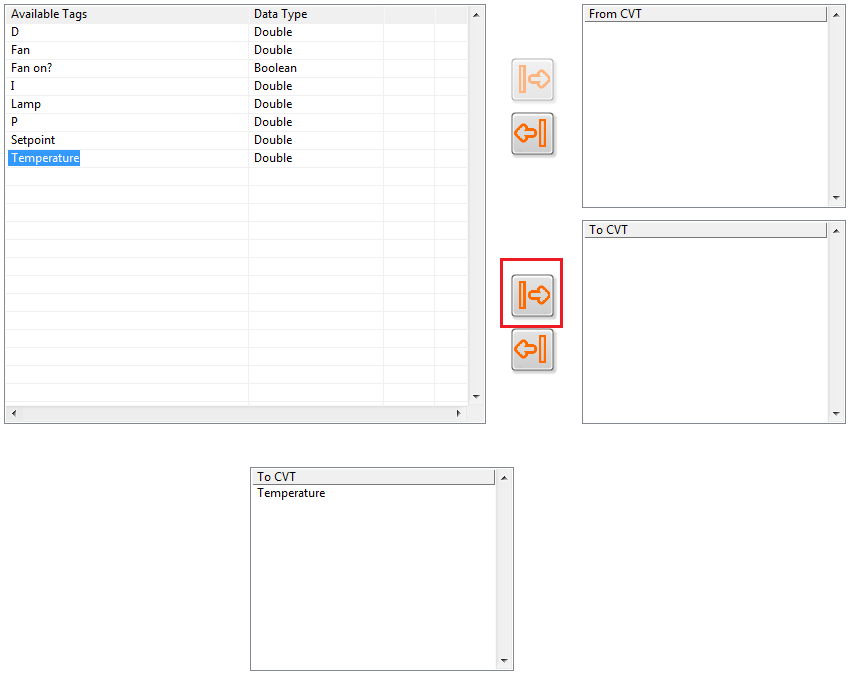
The template also creates an empty loop where application specific code can be added:

If a DCAF engine can't do something well, the functionality can still be added to the main application using standard LabVIEW programming practices. For example, the loop above could contain code to read a high-resolution stream of buffered data from an FPGA and stream that data over the network (something DCAF is not designed to do well). Meanwhile DCAF can still be used to handle the more standard control functionality.

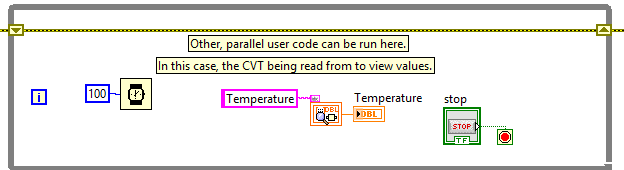
### Working outside of DCAF with CVT

When writing code in the main VI, it is often beneficial to have access to data within the engine. While there are many ways to do this, the most straightforward way to exchange data locally with an engine is to use the Current Value Table (CVT). CVT is a library that acts as a central data repository – more information can be found within the [CVT Library page](http://forums.ni.com/t5/Reference-Design-Content/LabVIEW-Current-Value-Table-CVT-Library/ta-p/3514251).

As an example, the DCAF Hands-On ([available on this page](https://github.com/LabVIEW-DCAF/Documentation/tree/master/Trunk/Hands%20On)) goes over adding a CVT tag to monitor temperature. The general process would be to bind an existing tag to a channel in the CVT module (image taken from the DCAF Hands on)



And then directly accessing that CVT Tag in the main application. Such as in the loop created for application specific code:



### Working outside DCAF – Shared Memory

Another possible way of passing DCAF data to code running in the same target is to use the Shared Memory Module. This Module allows a separate application running on a Linux system to exchange data with a DCAF engine running in a different built application.

[The Shared Memory Module Documentation can be found on the DCAF community page.](https://forums.ni.com/t5/Distributed-Control-Automation/DCAF-Shared-Memory-Module-Documentation/gpm-p/3620996)