Astemes Triarc Framework

Astemes - Anton Sundqvist

Anton Sundqvist

Table of contents

1. Introduction to Triarc Framework

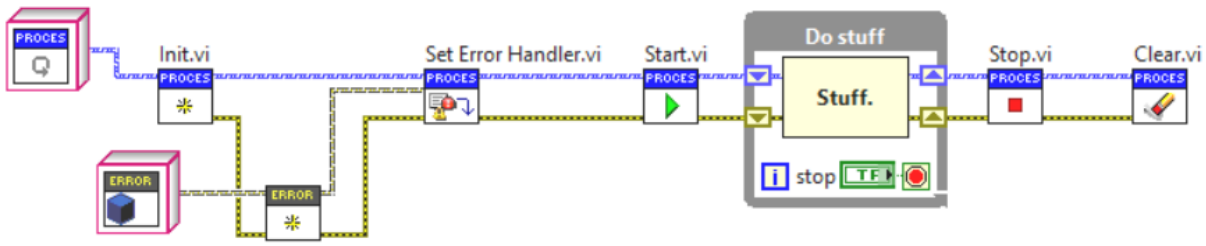
This document gives a brief introduction to and a high level overview of the Triarc Framework (`TF`). The framework is developed and maintained by Astemes and is released under and open source license.

1.1 What is Triarc Framework

The framework is an application framework useful for developing concurrent applications in LabVIEW Core abstractions for `Processes` , `Applications` , and `Views` are provided by the framework. These may be extended to implement sophisticated applications.

The framework provides communication channels between processes and manages their lifecycles. Boilerplate for error handling, logging and configuration management is also provided by the framework.

The API to a `TF` process is similar to the API of an instrument driver, and should for this reason feel familiar to any LabVIEW developer. An example is shown below, where the active lifecycle of the process would be during the while loop. From within the while loop, the user of the process would interract with it using API calls.



`TF` is based on the proven actor model, which is a very powerful way of managing distributed processing. The design goal of `TF` is to provide all the benefits of an actor based system, while maintaining a simple API and reducing unnecessary complexity.

The design is test driven, and because of this modules built on `TF` are testable and may be created to adhere to SOLID design principles.

1.2 Why Yet Another Framework?

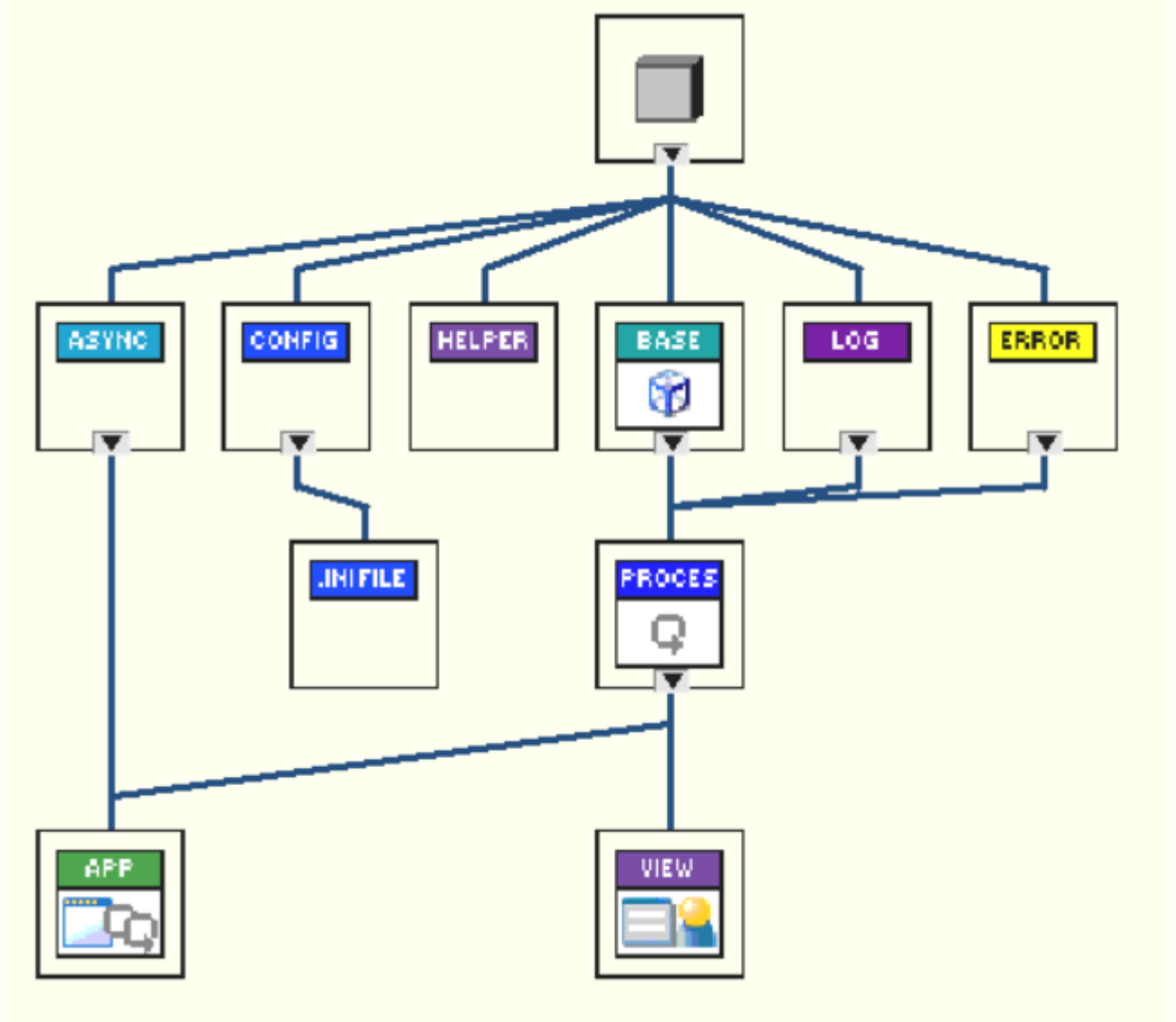
There are well established frameworks available in the LabVIEW ecosystem, most notably NI Actor Framework and DQMH. Each of these have their own benefits and drawbacks, but neither seemed to fit well for our needs and workflow. Thus a test rig architecture grew through and after a few incarnations, a generic high level framework could be extracted, to be further distilled, tested and improved.

Triarc processes are composable, independent, and testable. The framework encourages modularity and allows for composition of applications out of smaller single responsibility processes. Robust communication is implemented and each process life cycle is managed by the framework. Entry points are provided by the framework for error handling and logging, and a powerfull debugger is available.

Even if there might not be a great demand for a new framework, there was no reason to keep Triarc private or closed. By releasing it as open source, proper version management is enforced and publicly available. Releasing it as open source is valueable for all users of the framework and gives room for anyone to contribute both ideas and code.

2. Framework Architecture

This document describes the architecture and the dependencies between the core components of the Triarc Framework (`TF`). The hirarchy of the core classes and interfaces are shown below.



2.1 Framework classes

The framework classes are the starting point for every Triarc module. They are responsible for handling the lifecycle of the processes in the application as composable modules.

2.2 Framework Interfaces

The framework interfaces may be used to decorate a framework class and add additional behavior. As an example, adding a `Configuration File` interface to a class makes it configurable through the API defined by the interface.

3. Base

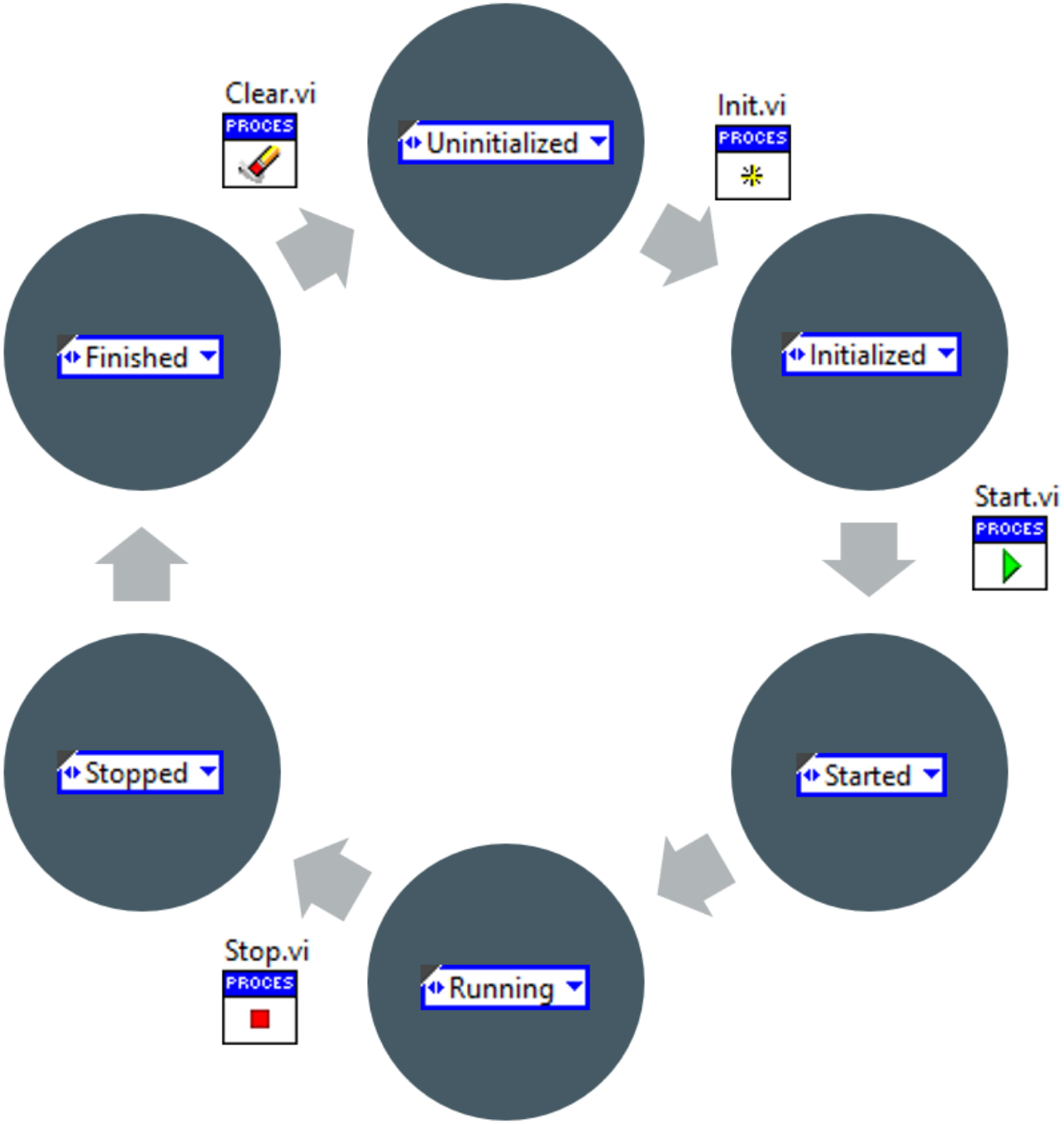
The `Base` class is a common ancestor for processes and may also be used to create components which do not run asynchronously. The `Base` class is responsible for storing the base name of the class, which is imutable and assigned on initialization. Additionally the class implements scaffolding which is used by the `Configuration Interface` methods.

4. Process

The `Process` is the fundamental building block of a Triarc application, similar to an actor in Actor Framework or a DQMH Module in DQMH.

4.1 Process Life Cycle

The framework maintains information on the lifecycle state of the process, which changes as the process changes state during execution of the application. The lifecycle state is a core concept which is used to ensure internal consistency of the framework. The normal evolution of the lifecycle is shown in the following image.



The process will always start as `Uninitialized` when created and move to the `Initialized` state when the `Init.vi` method is called. The `Init.vi` is a dynamic dispatch method which should be used to allocate any resources and open references used by the process. When the process is started, the main processing loop is launched asynchronously. The `Start.vi` method waits for the process to have handled the `Start` message and reached the `Running` state before resuming execution. The process lives its own life in the process loop until it is stopped by calling the `Stop.vi` method. The `Stop.vi` method also waits for the process to finish executing before resuming execution. Once finished, the process may either be restarted or cleared, by calling the `Clear.vi` to release any resources allocated during initialization.

The framework provides methods for reading the current lifecycle of the process or wait for a specific lifecycle state.

4.2 The Process Loop

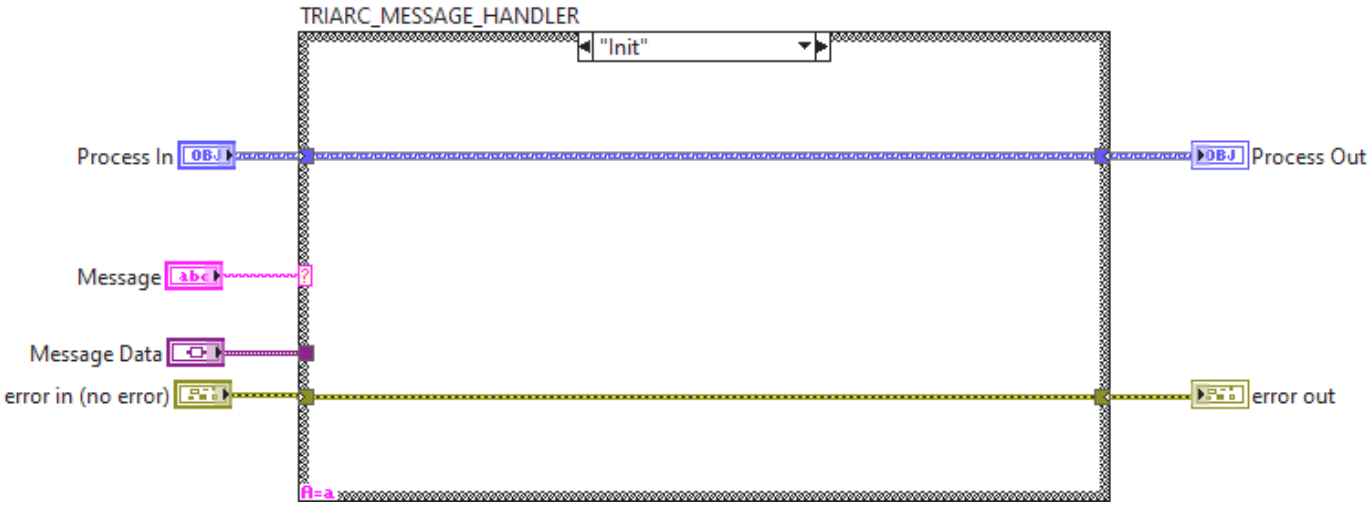
When a `Process` is started, the process loop is launched asynchronously. Each `Process` maintains its own state and state data in the Process Loop. A very important principle is that the state data may only be accessed from within the Process Loop. There is no limitation on the number of processes running concurrently and multiple processes of the same class may be started in parallell without limitations.

The process loop is esentially a queued message handler which receives messages enqueued by API-methods and passes them to a `Handle Messages.vi` method. The VI containing the process is private to the framework and may not be changed by a developer using the framework. To implement functionality into a process, the `Handle Messages.vi` is overridden.

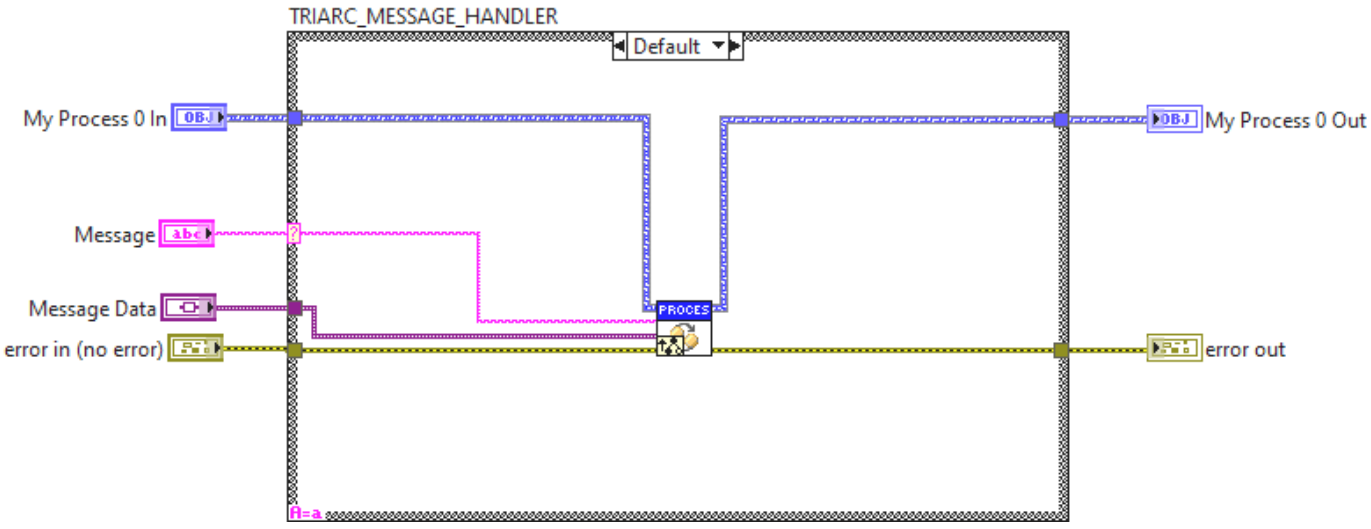
The framework implements functionality in the process for debugging, logging and error handling.

4.3 The `Handle Messages.vi`

To make a process do something interesting, the `Handle Messages.vi` should be overridden and implemented. The VI is called in the Process Loop whenever a message is received and by implementing the cases of the case structure, the behavior is defined. The process data should only be accessed within the `Handle Messages.vi` and this makes it safe to read and update the data without risks for race conditions.



The Default case of the case structure of the `Handle Messages.vi` should always call the `Call Parent Class Method.vi` to pass any unhandled message to the parent. This makes the message handling logic very DRY and only the specific responsibility of the `Process` is present in the message handler. As the number of cases in a `Handle Messages.vi` is typically rather small, the readability is often good.



4.4 The `Read Configuration.vi`

Configuration management is a central issue for most test systems, and for this reason configuration management is part of the framework. The `Read Configuration.vi` is a dynamic dispatch VI which may be overridden to load default configurations. The VI is called by the framework when the process is initialized.

4.5 Process Context

When a process is added to an application, the process is aware of its context. If process A is added to application B, then B is the context of A. If application B is further nested in applcation C, the context of A is still B, but the context of B is C. If application C is the top level application, it does not have a context.

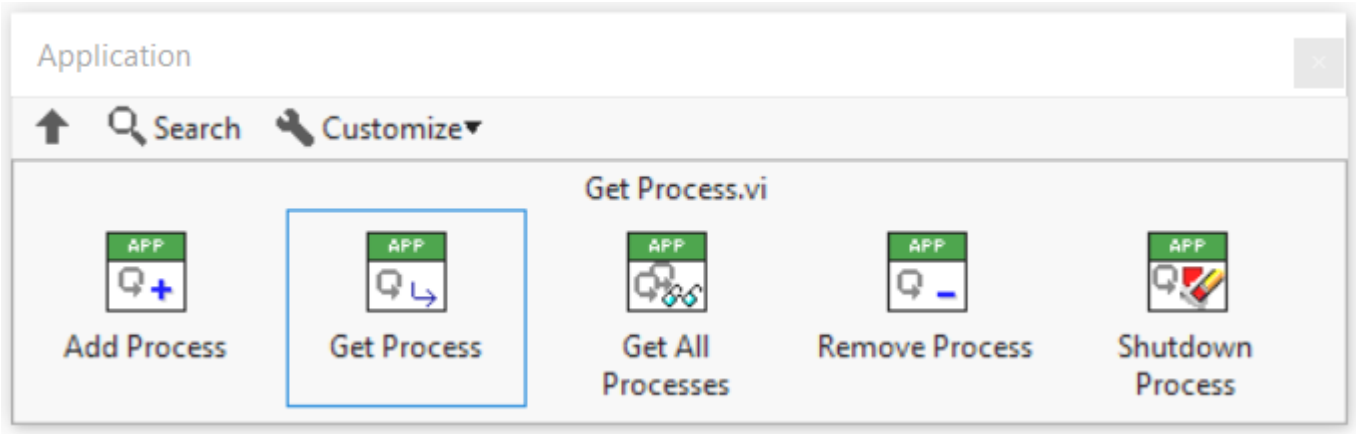
Tracking the context of a process is usefull as it allows for communicating from the process back to the caller. To make the communication useful, the process will need to make some assertions on the context in which it runs, *e.g.* that it implements a given interface.

5. Application

A computer program is typically composed of many small processes handling the different responsibilities and behaviors of the application. In Triarc `Processes` are composed in `Applications`.

5.1 The Triarc Application

The `Application` is a class derived from the `Process` class. The purpose of the `Application` is to manage the lifecycle of other processes. The `Application` has an API as shown below, and also inherits the API of the `Process`.



`Processes` are added to an applicaiton using the API VI called `Add Process.vi`, and may subsequently be retrieved from the `Application` using the `Get Process.vi`. Each `Process` can access its owning `Application` using the `Get Context.vi`.

As the `Application` is itself a `Process`, there is nothing preventing nesting applications. This enables creation of rich architectures with layers of sub systems.

5.2 Lifecycle Management

An application is responsible for its owned processes and the processes follow the lifecycle of the application. If the application is started or stopped, all its owned processes changes lifecycle states in the same way. Processes added while the application is running are not started automatically, which enables idle processes to be added to an application.

5.3 Recursion through the Application

The `Application` does apply certain operaitons recursively through its owned processes. This applies also in general for nested applications.

When a `Process` is added to an `Application` the lifecycle of the process is managed by the application. This means that if the owning `Application` is *e.g.* stopped, all `Processes` within the `Application` will also be stopped.

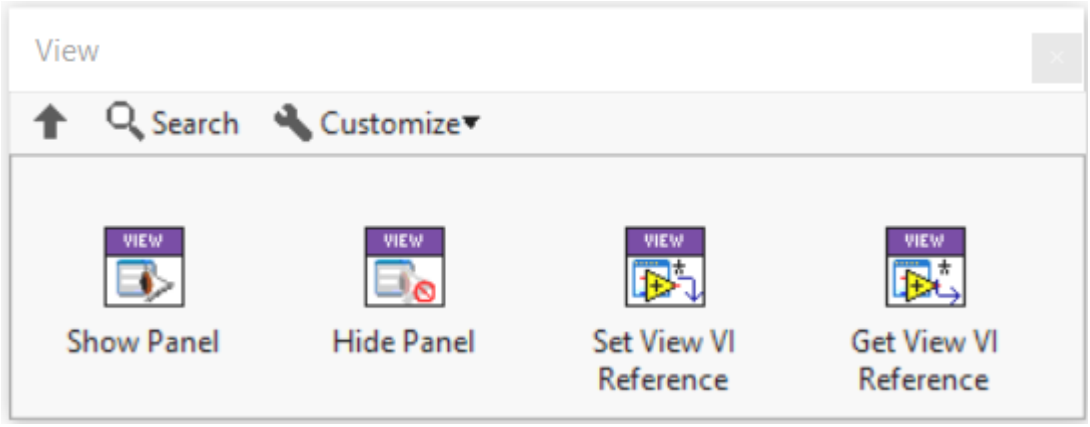
Similarly, configuring an `Error Handler` or a `Log Handler` for an `Application` will configure the handler for all owned `Processes`.

6. View

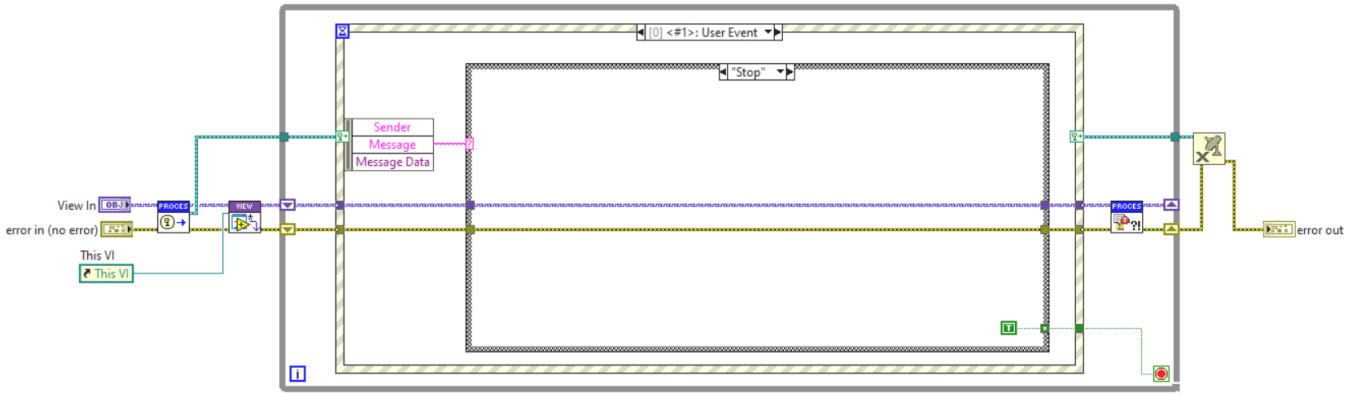
A Triarc `Process` is a headless process running without a user interface. While indicators may be added to the `Handle Messages.vi` for debugging, this is not recommended for implementing a user interface. The derived `View` class adds a user interface to the process. This VI should be used to implement a user interface component and complex user interfaces are typically composed from multiple small views inserted in sub panels.

6.1 Anatomy of a View

The `View` class adds a VI called `View.vi` and provides methods for showing and hiding the panel of the `View.vi`. The API also provides a method for getting a reference to the `View.vi`, which may be used to insert the `View` into a subpanel. The API is shown below.



There are two requiremetns that the framework asserts on the `View.vi` and it is important that these are fulfilled. The first is that the `View.vi` reports a refrence to the VI when it is started using the provided framework method. The second requirement is that the `View.vi` stops when the `Stop` message is broadcasted. This is demonstrated below. These requirements are tested in the unit test created with the class when the class is created from the `New Triarc Process` menu item.



6.2 Communicating between the Process and the View

The `View.vi` and `Handle Messages.vi` runs concurrently and in separate VIs with some given communcation channels.

When a user interacts with the view, the view may call any API method on the process to send messages to the process loop. As the `View.vi` belongs to the process, the API may even be privately scoped.

Communicating from hte `Handle Messages.vi` in the process loop to the `View.vi` is typically done using broadcasts. Broadcast messages are sent from the process loop and handled by the `View.vi` to update the user interface.

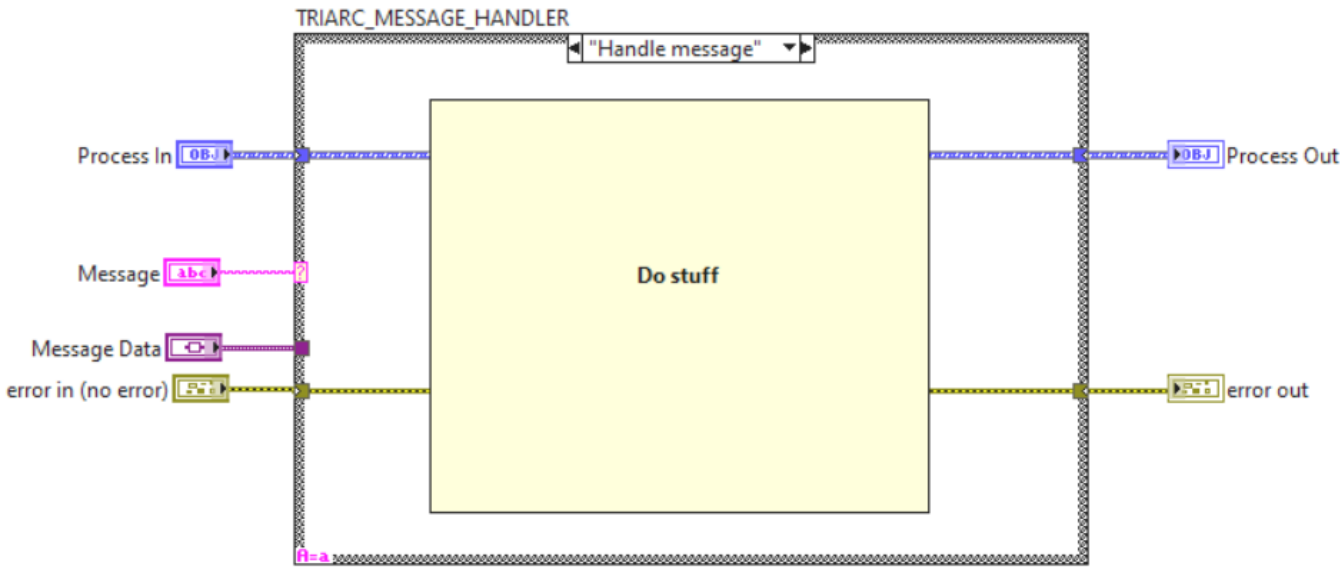
Another way of updating the user interface is to send references from the `View.vi` to indicators and then update these indicators using property nodes in the `Handle Messages.vi`.

7. Handling Messages

Messages are sent by users (not necessarily human users) of the process by making API calls on the process wire. All messages are enqueued with the same priority to a the process loop where they are received and fed to the `Handle Messages.vi`. This VI is the core of any process.

7.1 The `Handle Messages.vi`

The `Handle Message.vi` is called from within the process loop and receives every message sent to the process throug the API. It may be thought of as a subVI containing only the case structure in a traditional LabVIEW Queued Message Handler. As seen below, the `Handle Messages.vi` receives the message string and the message data variant and has access to the process wire.



A very important rule in Triarc is that the `Handle Messages.vi` is the only place where the data in the class wire is accessible and valid. In other words, this is the only place where the class wire may be unbundled or bundled.

7.2 Overriding Messages

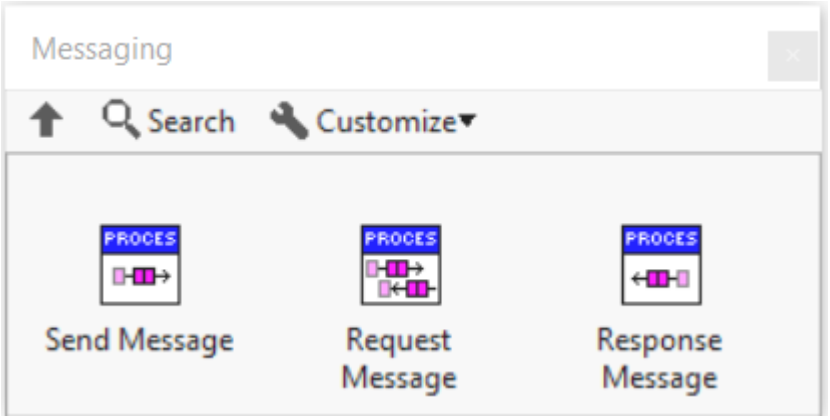
Since every message is passed to the `Handle Messages.vi`, it may also receive messages defiend by API calls in super classes of the specific process. The normal behavior is to invoke the `Call Parent Class Method.vi` in the default case and let the parent take care of the message. It is however possible to override the behavior and handle or add additional logic to API calls of parent processes. This is an example on how the Triarc process adheres to the open-closed principle.

8. Messaging

The main communication channel in Triarc is based on message passing to a process.

8.1 The Messaging API

The messaging API is relatively simple, as seen below.



There is no priority levels to be configured and messages are received in the order they are sent. Additionally, the method for receiving messages is private to the `Process` class and is only used in the process loop.

8.2 Sending Messages

A regular message is sent to the process using the `Send Message.vi` and the message is received in the Handle Messages VI of the process. Upon receiving a message the process may execute some logic, change its state data, launch new processes, broadcast a message, and so on.

In general the message handling should be atomic, implementing the full functionality in one of the cases of `Handle Messages.vi`. This is because there is no guarantee that other messages will not be interleaved if one API call sends several messages.

8.3 Messaging throught the Lifecycle

Messages may be sent both while a process is running and before it is started. Messages sent before starting a process will not be handled before starting the process.

When the process starts, the first message to be handled is always the framework defined `Init` message. After this all messages sent before starting the process is handled, and then the `Start` message is handled.

When stopping a process, a `Stop` message is sent. Messages sent after calling the `Stop.vi` method, but before the `Stop` message is handled will be handled before the process finishes. This is meant to allow for controlled and clean stopping of the process and if the process must be terminated quickly, the `Clear.vi` guarantees immediate termination.

If a process is stopped, it is still possible to send messages to it, but these will not be handled unless the process is started again. Restarting a process is a fully supported use case.

8.4 Type safety

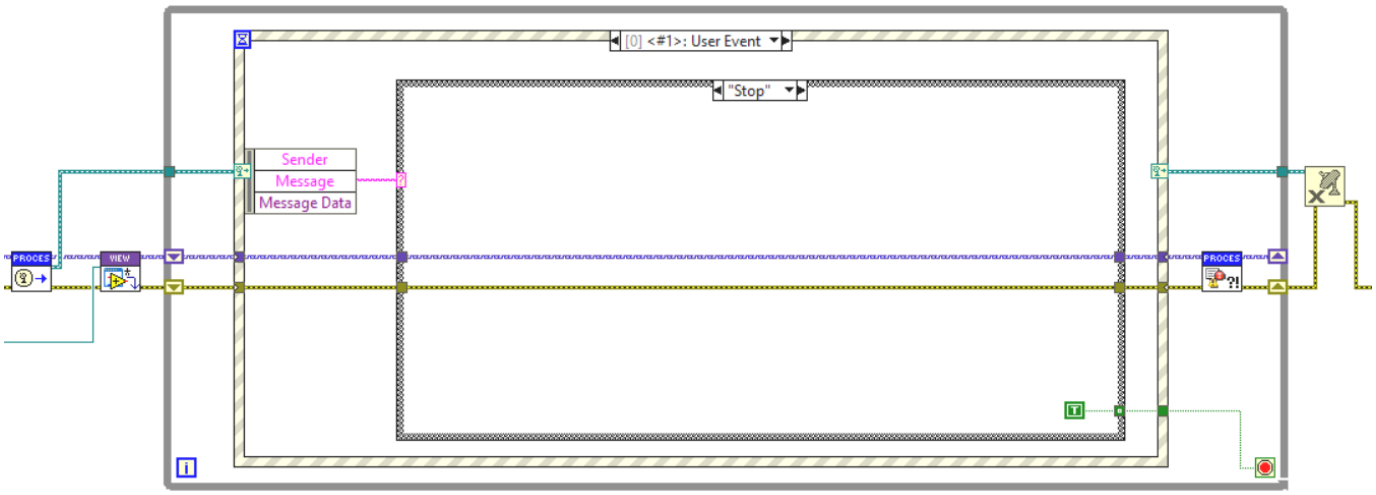
Message specification is restricted to the class handling the messages. The fact that the messaging VI:s are protected enforces the process to implement an API VI which enqueues the message and this VI should in general hard code the message string and enforce the expected data type for the message data. This pattern protects from misstyping the message string and ensures the expected data type is sent. It also enforces a clear and robust high-level API to be defined, which is useful for both testing and readability.

9. Broadcasting

Broadcasting is a communication channel used for sending one to many type of messages.

9.1 Implementation using User Events

The transport mechanism for broadcast messages is based on User Events with a fixed data type. The creation of the broadcast User Event reference and event registration is managed by framework methods. To register for broadcasts from a process, use the `Register for Broadcast Events.vi`. There is also a method available to recursively register for all processes within an `Application` called `Register for Broadcast Events Recursively.vi`.



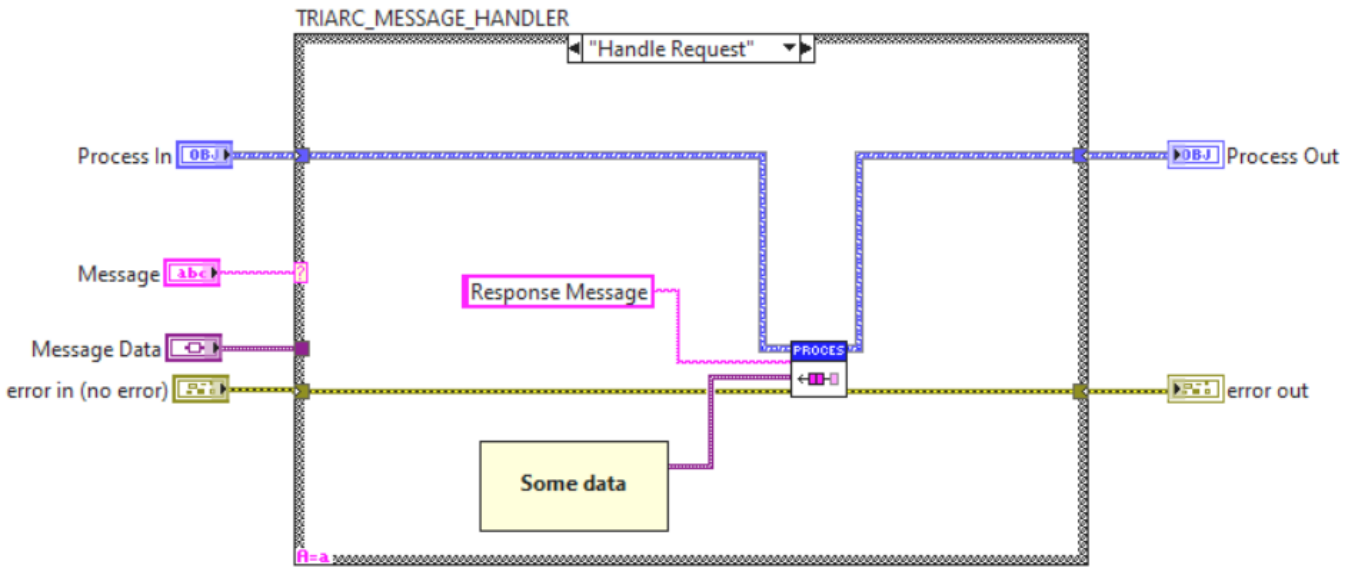
To receive broadcasts, an event structure should be used as shown above.

10. Requests and Responses

The Triarc framework consists of processes running independently which communicates through exchanging messages. The framework provides a broadcasting communication channel, which may be used to broadcast results from handling a message. In addition there is also the request message type which may be used to do blocking and non-blocking requests as described below.

10.1 Synchronous Requests and Response

Sometimes it is useful to make syncornous requests to a process. If something should be returned from the process to the sender of a message, the `Request Message.vi` may be used. The process receiving the request must call the `Response Message.vi` to return a response to the requestor and it must be called from the case in the Handle Messages VI which handles this specific message. This is shown in the image below.



Requesting something from a process is a serialized operation, meaning that the Request Message VI will block the execution until either a response is returned or the timeout has expired. If a request is made from within another process, this process will be stuck waiting for the reply and cannot respond to, or handle, any other messages while waiting for the response. This in turn introduces risks for deadlocks and other quirks of concurrent systems, so requests should be used with care.

The Triarc request-response pattern is a less tightly coupled solution because the request and response responsibility is implemented completely in the responding process class which exposes the request API call. Changing the implementation only requires changes to the responding class as long as the returned data type reamins unchanged. And if the type needs changing, it will cause a compile-time error which mitigates the risk of introducing bugs.

10.2 Asynchronous Requests

In some cases a synchronous request is acceptable, or even desireable, but there are many situations where a blocking wait for response is unacceptable. The recommended way of solving this in the Triarc framework is by using the `Async Request Message.vi` introduced in version 1.0.24. The `Async Request Message.vi` takes, in addition to the regular Message string and Message Data variant, a `Callback` interface. When the `Response Message.vi` is called from the process loop, the `Callback.vi` of the class, provided when calling `Async Request Message.vi`, is fired. This is a very powerful way of implementing requests, as it does not directly couple the callee to the caller. It is important to note that there is an indirect coupling, *i.e.* if data or formating of the response changes, it may breake the function of caller.

10.3 Asynchronous Requests prior to Version 1.0.24

An asynchronous action runs in parallell to the process and is useful for dispatching requests without blocking the process. To implement an asynchronous request, the request API VI is placed in the Helper Action VI, and when the request needs to be dispatched, the Launch Asynchronous Action VI is called. The request is in this scenario running assynchronously to the process and it is even possible to have the same process dispatch several concurrent asynchronous actions waiting for different requests.

When the response eventually returns, the response message is received in the Async Action VI and may be forwarded to the main process as regular message or a process internal message. It is important to note that the asynchronous action is outside the process loop and cannot directly change the state of the process, but it may enqueue messages to the process which updates the state. Another caveat is that a change in state in the process will not be propagated to a running action. In order to propagate any state chnage to a running asynchronous action, it must be restarted from within the process loop. An important virtue of this implementation is that the protection of the message queue is preserved and the requesting class decides what message is sent to its process when the response is returned.

11. Error Handling

Error handling is important for building reliable systems. Triarc provides very flexible error handling mechanism by delegating error handling to a user specified error handler.

11.1 Implementing and Error Handler

To implement an error handler, it is necessary to create a class which implements the `Error Handler Interface` provided by the framework. The interface requires the implementer to override the `Handle Errors.vi` to implement the error handling logic.

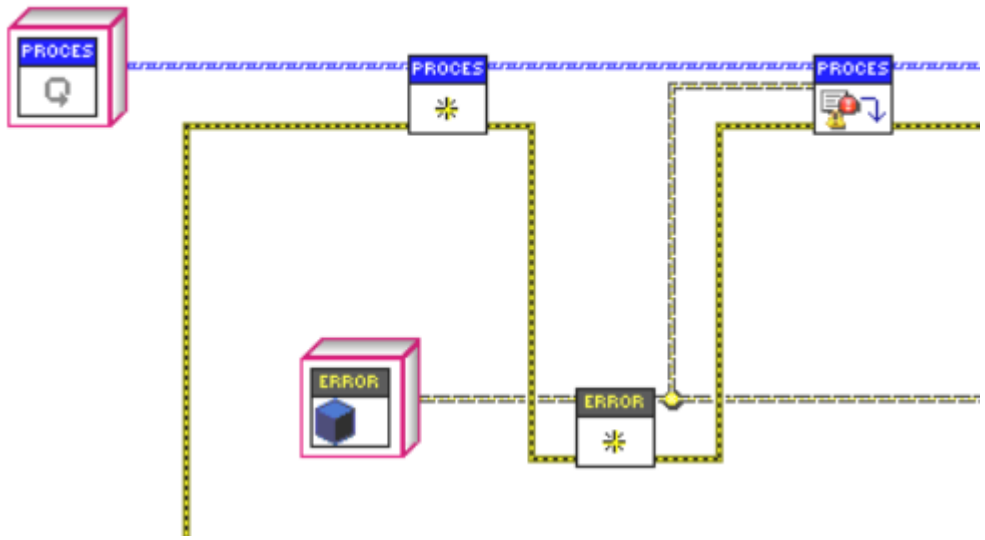
11.2 Error Handling in the Process

The Process class also derives from the `Error Handler Interface`, but the actual handling of the error is delegated to the configured error handler. In this way, error handling may be invoked by calling the `Handle Errors.vi` on any process wire.

The `Location` input to the `Handle Errors.vi` is used to specify where the error occured. When an error occurs within the `Handle Messages.vi` of a process, the location contains the qualified name of the process followed by the message handled while encountering the error.

11.3 Setting an Error Handler

An error handler may be configured for a process using hte `Set Error Handler` VI. This works recursively for processes within an application, regardless of wheter the process was added to the application before or after the error handler was configured.



After setting the error handler, any error occuring within the process loop will be delegated to the `Handle Errors.vi` of the configured error handler. This makes it possible for an error handler, which *e.g.* displays any occuring error while testing, to be replaced by something more appropriate for production.

11.4 Error Handling in an Application

When the `Handle Errors.vi` of a Process is called, the error handling is delegated to the error handler configured using the `Set Error Handler.vi`. If an error handler has not been set for the process, the error handling is delegated to the context of the process. This means that the error is passed to the `Application` which owns the Process and delegated to its configured error handler. This happens recursively through the process tree and the first context which has an error handler configured will handle the error. This means that it is enough to configure the error handler only for the top level `Application` and all processes in the hierarchy will delegate error handling to this error handler. If a certain process needs to override the error handling, another error handler may be configured for this process and *e.g.* supervisors may be implemented.

11.5 Logging Errors

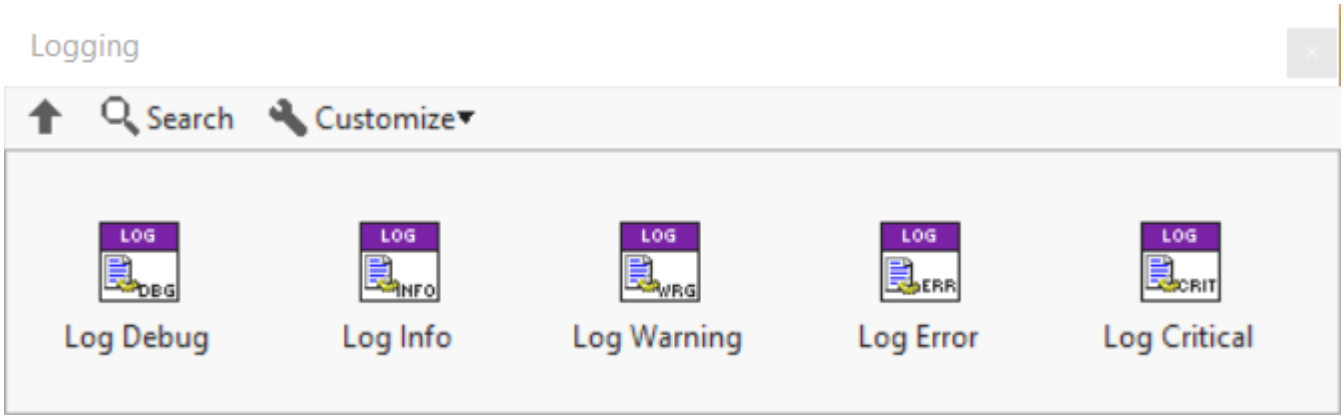
The concepts of error handling and logging are separated in Triarc and an error handler should in most cases not be responsible for logging errors. The responsibility of logging should generally be implemented by a log handler, as it is often useful to know the context of errors occuring.

12. Logging

Logging is very useful for gaining insight into a running system, both during development and when deployed. Most modern programming languages provides the scaffolding for writing log information, but LabVIEW does not. Triarc provides a powerful and flexible logging mechanism.

12.1 Logging API

The logging API may be used with any Triarc Process and consists of VIs for logging messages with different severity levels.



The Debug severity only logs if a conditionally disabled symbol `DEBUG=TRUE` is configured for the project, and should be used when debugging issues and logging detailed data. The logged message will contain data on the process which called it.

12.2 Framework events

The framework calls the logging API at certain events, such as starting and stopping processes. These events are sent to the logger at the `INFO` severity level. Errors and warningsa are sent to the logger by the framework as they occur and these will occur with the `WARNING` or `ERROR` severity levels.

12.3 Handling Log Messages

What to be done with logged messages is up to the user of the framework. The framework delegates the actual logging to a class implemeting the `Logging` interface. The class handling the logging is configured using the `Add Log Handler.vi` and it is possible to have multiple log handlers configured concurrently. The log handlers are configured recursively for processes within an application.

12.4 Implementing a Log Handler

To implement a log handler, a class must be created which implements the `Logging` interface. There is one method called `Log.vi` which must be overridden to define what to do with the event to be logged. Common implementations is to display the log in a text indicator or log the content to a file.

To see a basic demonstration of a log handler, see the Coffee Shop example application.

12.5 Logging Severity

There are five levels of severity used by the logging framework. These are available through static VI:s in the `Logging` interface and are called `Log Debug.vi` , `Log Info.vi` , `Log Warning.vi` , `Log Error.vi` , and `Log Critical.vi` .

The Triarc Process implements the logging functionality by delegating to the configured log handler. The events are filtered by the process class and a log handler should only log the different levels of events without further filtering. The meaning of the levels in the context of a Triarc Process is as follows.

- `DEBUG` - Event is only logged if the `DEBUG==True` conditional disabled symbol is configured in the project.
- `INFO` - Events containing information that the application is running as expected. If an error occurs upstreams from the `Log Info.vi`, the logging will be ignored.
- `WARNING` - This event is invoked if a warning (error cluster with status == False and non-zero error code) occurs within the process.
- `ERROR` - Invoked when an error occurs in the process and should be used to log errors.
- `CRITICAL` - Invoked if called with an uppstreams error. Not used by the framework.

12.6 Framework Log Events

If the `DEBUG` conditional disabled symbol is set to `True` the framework will log every message in the order they are received.

The framework passes warnings and errors occuring in the process loop to the configured log handlers.

Additionally the following life cycle changes are logged as `INFO` level messages.

- `Started` - Log entry made when process started
- `Running` - Log entry made when process has executed the `Start` state
- `Stopped` - Log entry made when process stopped
- `Finished` - Log entry made when process finished

These events are useful as it allows for determination of wether errors and other events occurs during startup, execution on shutdown of the process.

13. Helper Loops

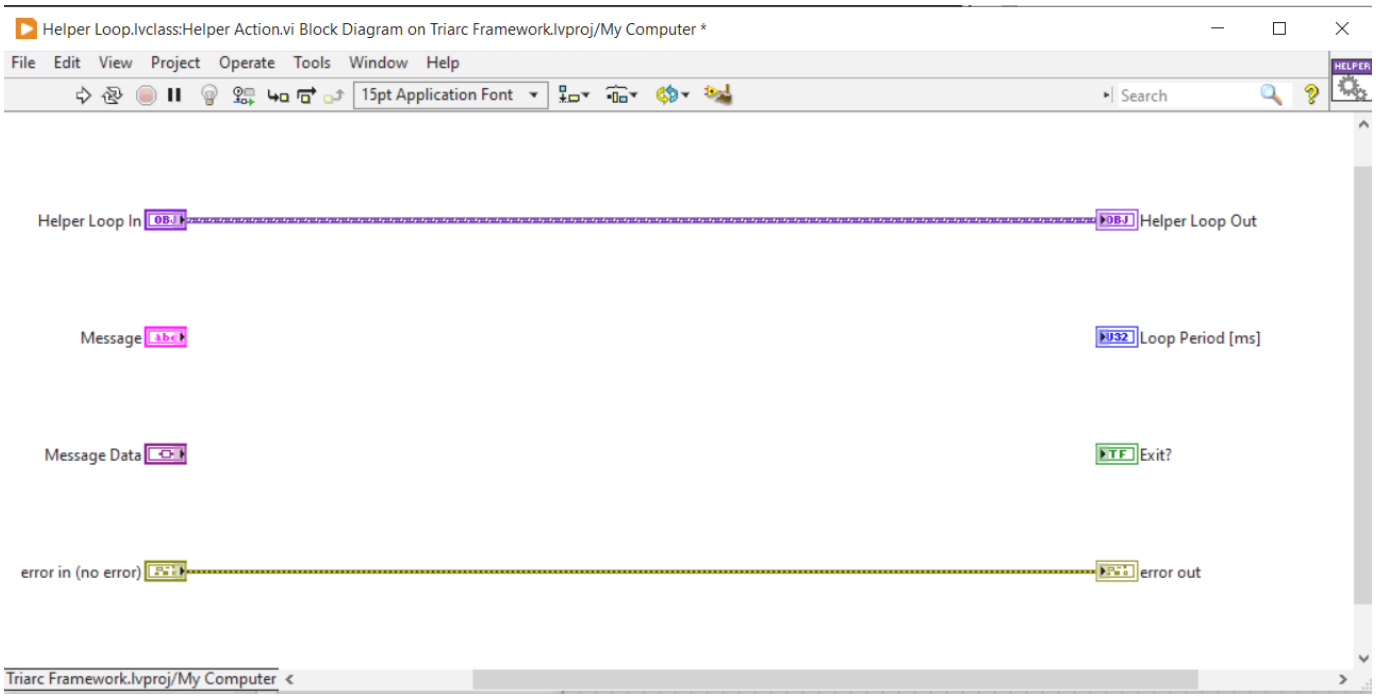
A common pattern in LabVIEW is the useage of helper loops to handle periodic tasks. Triarc provides a generic implementation for running helper loops and managing the lifecycle.

13.1 Use cases

There are often use cases where a task needs to be done periodically. Examples of this could be making a software timed measurement, reading from a buffer, or reconnecting to a network service. This could in principle be solved by re-sending the same message over and over to the process, but doing so has drawbacks and timing limitations. A better way of solving this is to use a helper loop running in parallell to the process.

13.2 Using a Helper Loop

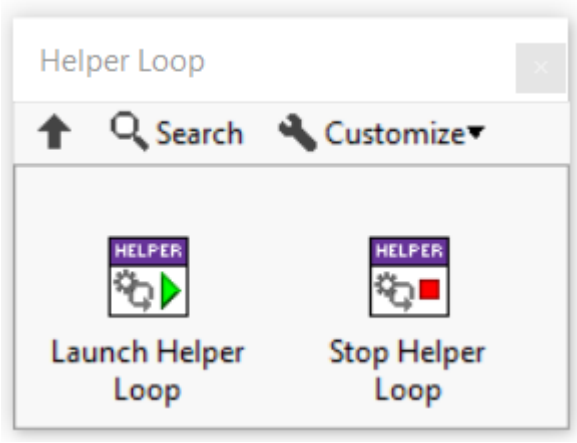
To add one or many helper loops to a process, the host process should be configured to implement the `Helper Loop` interface. The interface requires the implementation of a dynamic dispatch VI called `Helper Action.vi` which defines the action taken each iteration of the helper loop. The `Helper Action.vi` is shown below.



When a helper loop is started, the `Helper Action.vi` will be called periodically with the period specified by the `Loop Period [ms]` indicator. The loop runs until the `Exit?` indicator is set to `True` or an error is passed out of the `Helper Action.vi`.

13.3 Launching a Helper Loop

To launch a helper loop from within the host process `Handle Messages.vi`, one simply calls the `Launch Helper Loop.vi`. It is possible for one host process to have many helper loops running simultaneously. The `Launch Helper Loop.vi` accepts a Message string and a Message Data variant, which may be used to paramterize helper loops and alter the helper behavior.



All helper loops are stopped by the framework when the process finishes or is cleared. It is also possible to stop a running helper loop by calling the `Stop Helper Loops.vi`, shown in the palette above. Note that VI stops all running helper loops, and there is no way to selectively stop helper loops provided by the framework.

13.4 State Data

The `Helper Action.vi` has the class data flowing through it and values within the class data contains a snapshot of the state of the host process when the helper loop was started. That means that the data in the helper loop will not be updated when the state of the host process changes, and if the helper loop needs access to updated data it must be restared.

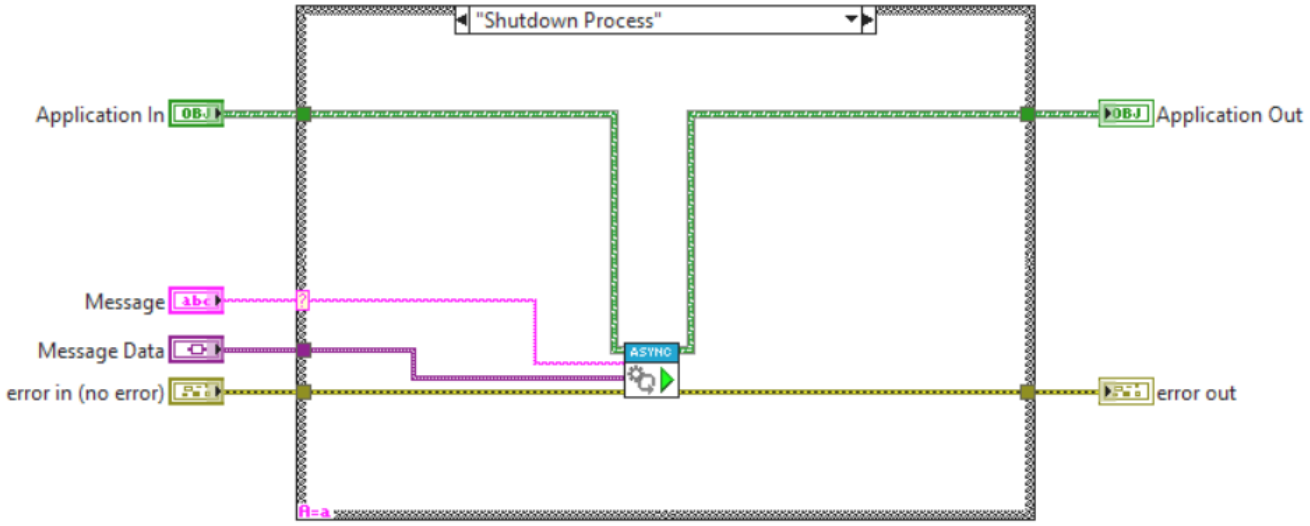
Similarly, the state of the process cannot be changed from within the helper loop. If data needs to be communicated back to the host process, regular API calls should be used.

14. Async Handler

Some messages may take a significant amount of time to process and it may be desireable for the process to remain responsive whild waiting for the handling of the message to finish. In such cases, an Async Handler is useful as it makes it possible to handle the message asynchronously while the process remains responsive.

14.1 Using an Async Handler

To add asynchronous handling to a process, the host process should implement the `Async Handler` and override the `Async Action.vi` to define an action. To launch the asynchronous handler, call the `Launch Async Handler.vi` . An example is shown below where the `Application` class lauches an asynchronous handler to shutdown a process.

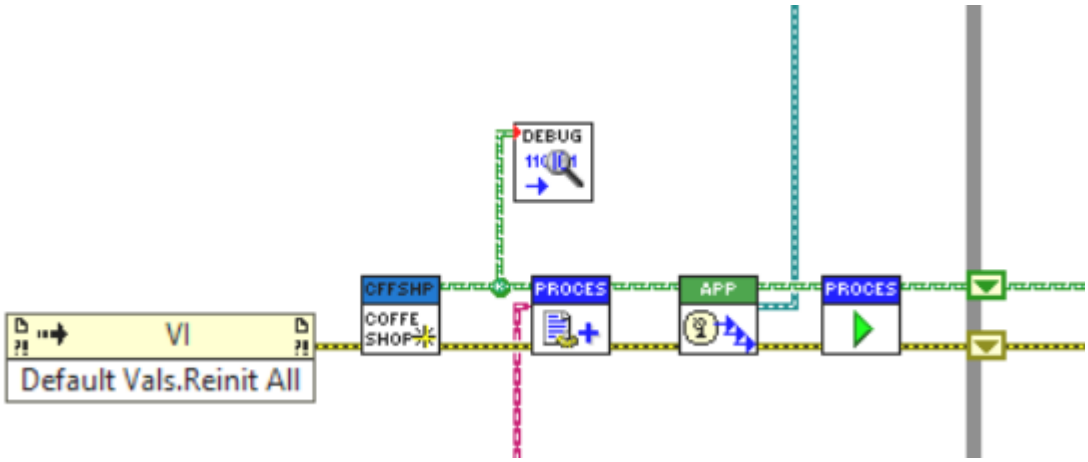


15. Debugging

Triarc comes with a powerful debugger which enables viewing of messages in an application in real time. The debugger is maintained in its own repository and deployed in a separate VI Package which must be installed in order to use it. While the debugger is an invaluable tool, overusing it should be avoided in favor of automated testing.

15.1 Using the Debugger

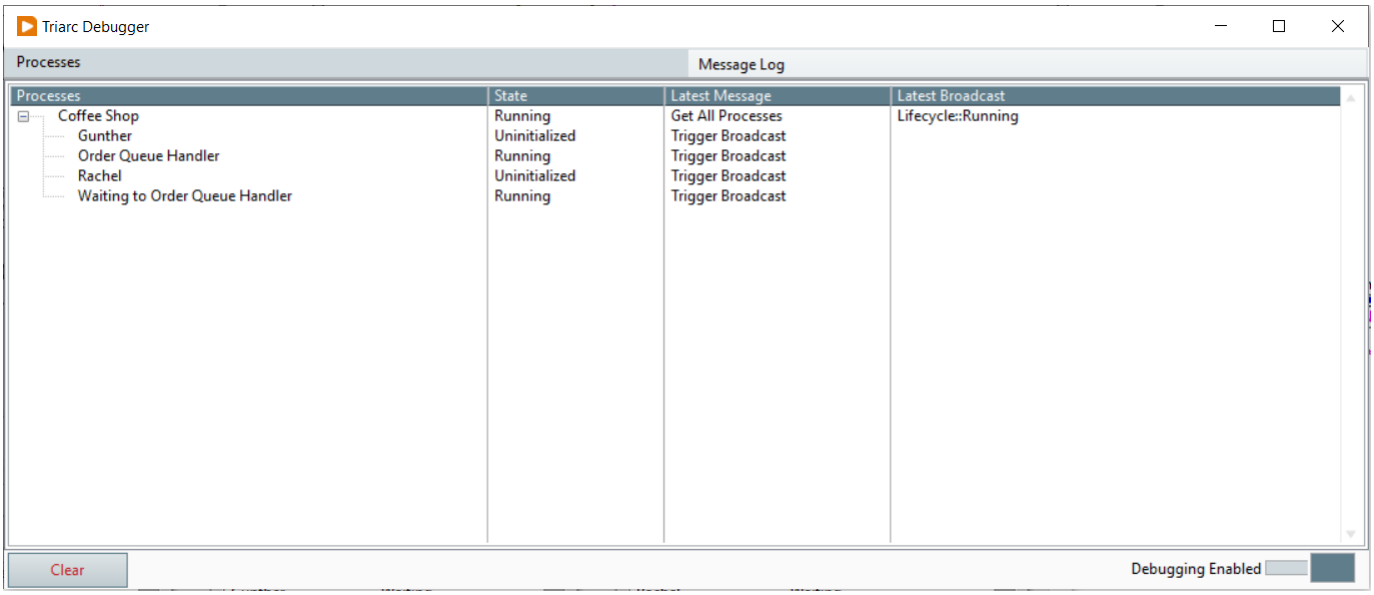
To launch the debugger, in LabVIEW, select Tools -> Triarc -> Debugger.. and the Debugger window will load. The debugger may also be launched by attaching it to the top level application, as shown below. The `Attach Debugger.vi` is found in the palette under `Triarc Debugger`. When the debugger is attached, it will also enable viewing of broadcasts from processes within the application.



To enable debugging, a `debug = TRUE` conditional disabled flag need to be set for the project. This flag may be set directly from the debugger window using the switch in the lower right corner. It is recommended to turn debugging of for deployment as the log level of the framework is increased and debugging adds a slight performance overhead.

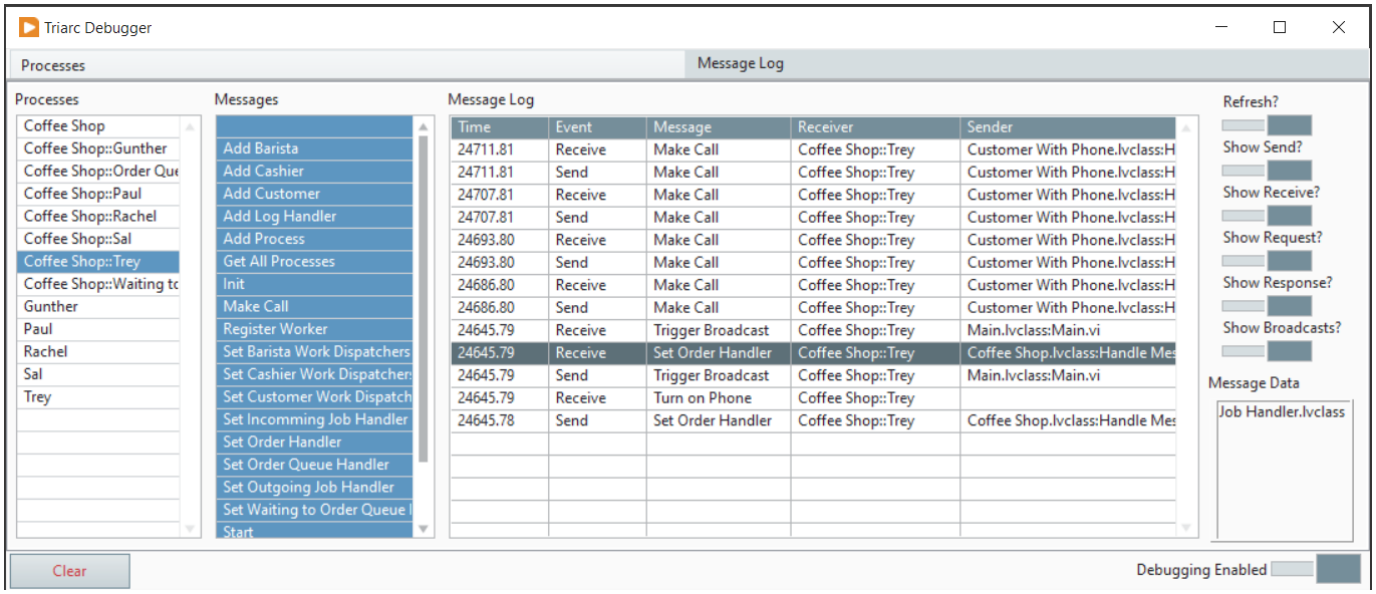
15.2 Process View

The Process View of shows all active processes, their state and the latest messages received. By double clicking on a process, the `Handle Messages.vi` of the process is opened and may be debugged using regular LabVIEW debugging tools. If the debugger was attached to the process or its owning application, broadcast messages will also be shown.



15.3 Message View

The Message View shows a log of messages sent in the application in real time. The message list may be filtered by process and by message, and further to only show *e.g* sent or received messages. By selecting a message, the content of the Message Data variant may be seen in teh Message Data indicator. Double clicking on a sent message will open the location the message was sent from and double clicking on a received message will open the receiving `Handle Messages.vi`.



16. Configuration Management

Configuration management is common in most software system and test systems are no exception, quite the contrary. It is useful to separate out parameters, which are to be editable after the system has been built and deployed, and manage these parameters outside the source. Because configuration management is so central to most test systems, the Triarc Framework provides a framework for managing configurations.

Configurations may be stored in any conceivable format, and one common implementation is to use human readable text files. LabVIEW comes with built in functions for editing key-value pairs in windows style `.ini` text files. Another common practice is to use the windows registry or a database for storing configurations.

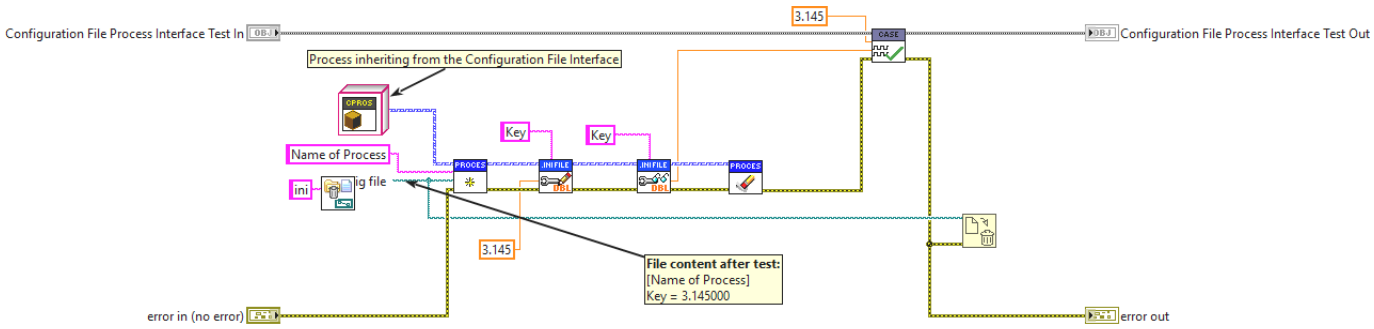
16.1 Triarc Configuration Interface

The Configuration interface defines abstract methods for writing and reading configuration data. The implementation may be using any persistance technology. This is useful, as an in-memory implementation may be used for testing purposes of as a property object carrying configuration data.

16.2 Triarc Configuration File Interface

A common way of persisting configurations is to use human readable flat text files. This works well in many cases and text files are easy to manage and version control.

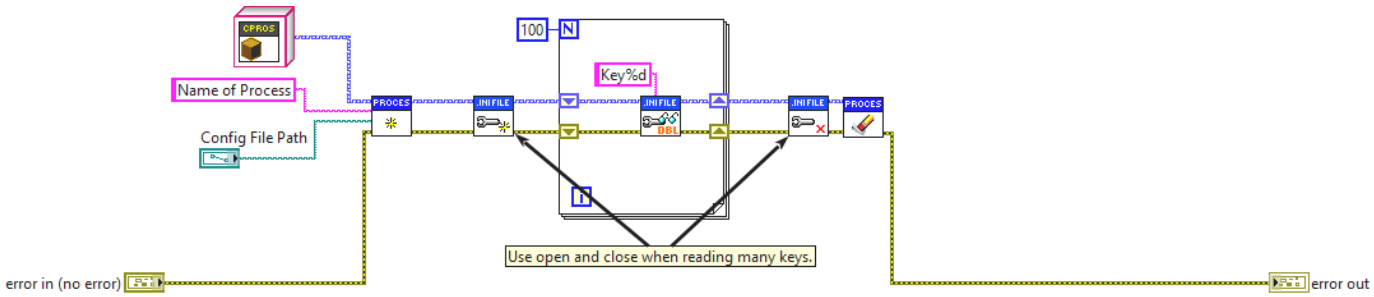
Triarc Framework provides the boiler plate for using configuration files to manage configurations. The API is shown in the test case below and the configurations are written to the text file.



The Base.lvclass implements the boiler plate code for carrying the reference to the file.

The Configuration File API implementation is designed to be thread safe and is optimized for reading. This means that many processes may read from and write to the same configuration file without race conditions. This is done by only allowing one process to write to the configuration file at the same time.

If many keys are to be read from the configuration file sequentially, the Open Config.vi and Close Config.vi methods shoule be used to reduce over head of opening and closing the reference for each read, as shown below.



16.3 Practices and antipatterns

There are a few pitfalls to be aware of when working with configurations and these are often overlooked. The following section assumes that configurations are stored in a file, but the same applies regardless of storage technology.

16.3.1 Configuration Data is Global

As configurations are stored in one (or potentially many) text file, any part of the applications which can find the file and parse it will be able to access the configuration information. This implies that there is an inherit risk for race conditions and care should be taken whenever a configuration is updated.

In `TF` the configuration management functions ensures that each process can only access its own section of the configuration file. In this way the risk for race conditions are reduced.

16.3.2 Race Conditions

If a reference is opened for writing in many multiple places concurrently, there is a risk for race conditions to occur. This is as the last process which closes the file will overwrite the previous writes since it opened the reference to the file. As mentioned earlier, this is handled by `TF` as synchronization mechanisms is implemented to only allow a single process to open the file for writing.

16.3.3 Default Values

If a key is not found, the LabVIEW native configuration file functions returns a default value, which may be set by the developer. If a key is *e.g.* misspelled in the file, a user of the system will not get any kind of feedback on what the system is expecting from the configuration file and may cause a lot of pain.

To mittigate this, `TF` will write the default value to the configuration file if the key was not found. By doing this, missing keys will be added to the file and if the file is missing all together a file with default configurations will be created automatically.

16.3.4 Lack of structure

Using LabVIEW native configuration file methods puts the responsibility on the developer to use reasonable sections when saving configuration data. This often leads to a structure that degrades over time as more fields are added.

`TF` enforces a structure on the configuration file, as data is modelled in such a way that each process has its own section. If, as an example, two instances of a process are used to interract with two hardware devices, each instance will have its own section named after the process. In this way the two hardware devices can be configured independently and the only action needed by the developer is to ensure that they are named uniquely.

17. Best Practices

17.1 Don't Marry the Framework

17.2 A good Process is an Idle Process

To ensure responsiveness of the system, processes should spend most of their time waiting for messages. To enable this, heavy processing should be dispatched into a `Helper Loop` and operations waiting with a long timeout should be handled by an `Async Action`.

17.3 Avoid sending messages to self

17.4 Use helper loops for repetitive tasks

17.5 Use async handler for blocking tasks

17.6 Separate core from framework

18. Design Discussion

When designing the Triarc Framework, one of the goals was approachability and ease of use while retaining the full power of the actor model. Software design is always about making trade offs, and in order to make the readability higher some trade offs have been made. This document discusses the most important of theses.

18.1 Values and Design Principles

The core values and design principles employed by `TF` are discussed in this section.

18.1.1 Let the data flow

The concept of data flow is one of the key strengths of LabVIEW and should be embraced. Passing data by value is to prefer as default, over passing data by reference.

18.1.2 Processing in the Process

All major processing should take place within the process. This makes API calls fast and reliable, while protecting the consitency of the process state.

18.1.3 Messaging

Information is exchanged between processes using asynchronous messaging. All messages have equal priority, to reduce unnecessary complexity. The handling of each message should be atomic, *i.e* should not be spread out over multiple states.

18.1.4 No Global Mutable State

By avoiding any kind of global mutable state, applications become more robust and easier to understand and reason about. It is also a prerequisite for effective testing, as tests would interfere if they were to share global resources. This means that global variables, functional global variables (FGVs), named queueus, etc. are prohibited.

18.1.5 Strict Typing

By using strictly typed APIs, errors can be detected at compile time and the consequence of changes to a data type is clearer. API:s should therefore use strict typing whenever possible.

18.1.6 Open to Extenssion

Classes are to be open for extenssion without modifying the base, while keeping the base behaviour unchanged. Triarc provides mechanisms for overriding behavior by either overriding dynamic dispatch VI:s or by handling messages defined by parent processes.

18.1.7 Everything in its Own Context

The lifecycle of every process running in an application is managed by the owning context. What is initialized must be cleared. What is started must be stopped.

18.1.8 Testing over Debugging

While the framework comes with a capable debugger, using it should however be relatively rare. Good test coverage is always to prefer and it is much more effective to debug a limited test case than a complete application.

18.2 Separation of Enqueuer and Actor

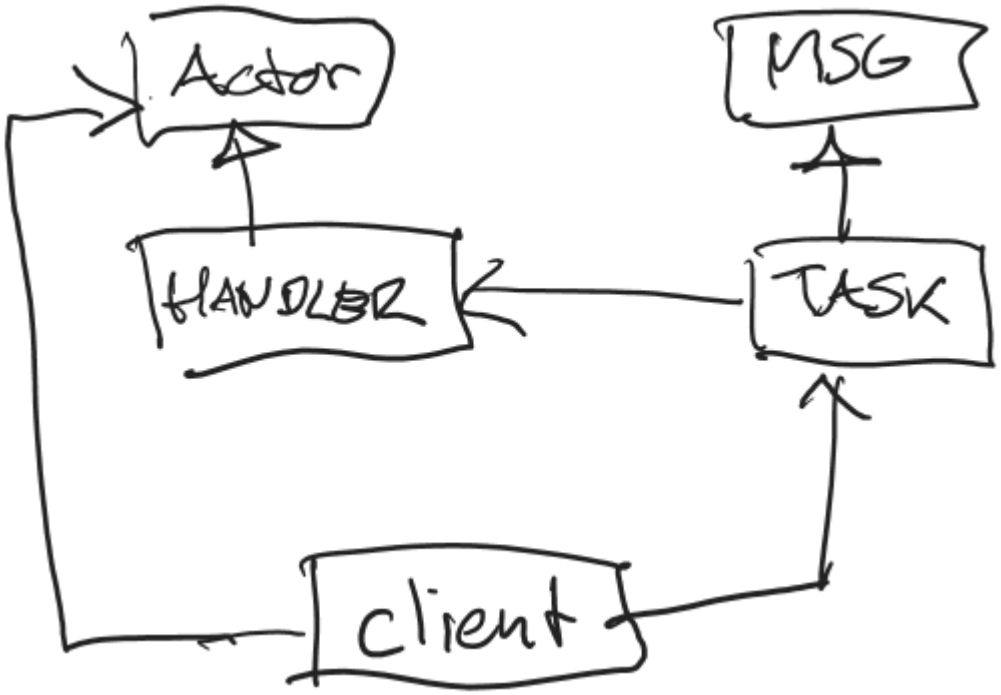
The main design difference between Actor Framework (AF) and Triarc Framework (TF) is how the enqueuer concept is implemented. In AF messages are sent by calling VI:s on the enqueuer object associated with the actor. This is done to protect the state data of the actor, by making it impossible to access it from the enqueuer object.

In TF the enqueuer and process classes are deliberately not separated from each other. This puts the responsibility on the developer to not access the data outside the Handle Messages.vi. Utilities are provided to determine whether a VI is called within the process or not, and in practice it is not difficult.

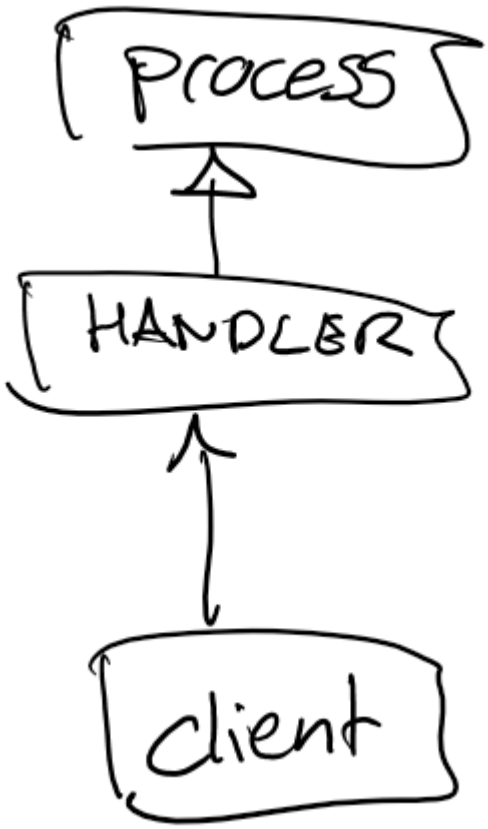
18.3 Orders of Complexity

The reason why the enqueuer is not extracted as a separate class or interface in TF is because it comes with a cost. Doing it introduces a lot of complexity as in the AF case messages needs to be defined in separate classes, nesting calls to public methods on the receiving actor.

Making an AF Actor called Actor A do something requires the caller to determine that the `something` message is compatible with Actor A. If the message calles a dynamic dispatch method on a parent of Actor A, the Actor A may override this VI to change what something it does. The call structure would look something like the diagram below.



In TF the corresponding action is simply to call the `something` API VI on the Process. If the process cannot do `something` the run arrow will be broken. The call hierarcy is much simpler, as shown below.



18.4 Semantics

There is a slight but important semantic difference in how you interact with an AF Actor compared to a TF Process.

When you want an AF Actor to do something, you enqueue a message to the actor using its enqueuer. You should know that the actor can handle the message you send it, or you will get a nasty run-time error. There is thus an implicit coupling between the actor and the message, even if there might not be any source code coupling from the actor to the message.

When you want a TF Process to do something, you make an API call on the process object. Under the hood, the API call enqueues a message to the Process, but the coupling is made explicit.

18.5 Priority Queues

Some frameworks give the option of sending messages with different priority levels. This adds significant complexity to the application since it increases the number of possible paths through the application. There are situations where a different priority message is usefull, but they should be rare.

Triarc provides the option of terminating a process by clearing it, so there is no need to send a high priority shutdown message. If it is not an emergency shutdown, it is in most cases better to shut down normally by stopping it and allowing each process to perform potential clean-up.

19. Triarc and Actor Framework

LabVIEW has already existed for over 30 years and Triarc is not the first framework which has been proposed. Comparing frameworks is not a simple task as different frameworks have merits in different areas. There is simply no such thing as the best framework, it all comes down to who you ask. As of today, there are two frameworks which have achieved a somewhat broad user base. The first is NI Actor Framework is bundled with LabVIEW and is included in the installation of LabVIEW. The second is Delacor DQMH, which is an extenssion and improvement of the NI Queued Message Handler (QMH) template.

Disclaimer: I tend to be strongly opinionated and I am certainly biased, as I would not have created Triarc if I thought there was better alternatives out there for me.

19.1 Brief Introduction to Actor Framework

The Actor Framework is bundled with LabVIEW since LabVIEW 2012 and is maintained by NI. It implements an actor model with independent actors passing messages to each other asynchronously. Each actor is a class overriding the base Actor class and multiple instances of the same actor may be instantiated.

Actors communicates by sending messages, which are objects. The Do method of the message defines the action taken by the actor when the message is received and in contains the functionality of the actor. The actor is fed as an input to the Do method of the message and may implement VIs used by the Do method.

As actors are classes, it is possible to create hierarchies of actors. Each actor is able to handle its own messages, as well as messages defined for any of its parents. The parents may allow child actors to override methods used in its messages to override the behavior.

From my own experience, actor framework is regarded as a powerful, yet very complex framework. I would recommend at least a CLA level of LabVIEW experience before trying to use actor framework.

19.2 Comparing Actor Framework to Triarc

Both Triarc and Actor Framework implements a version of the actor model and are for this reason very similar conceptually. In Triarc the actors are however called processes to avoid confusion.

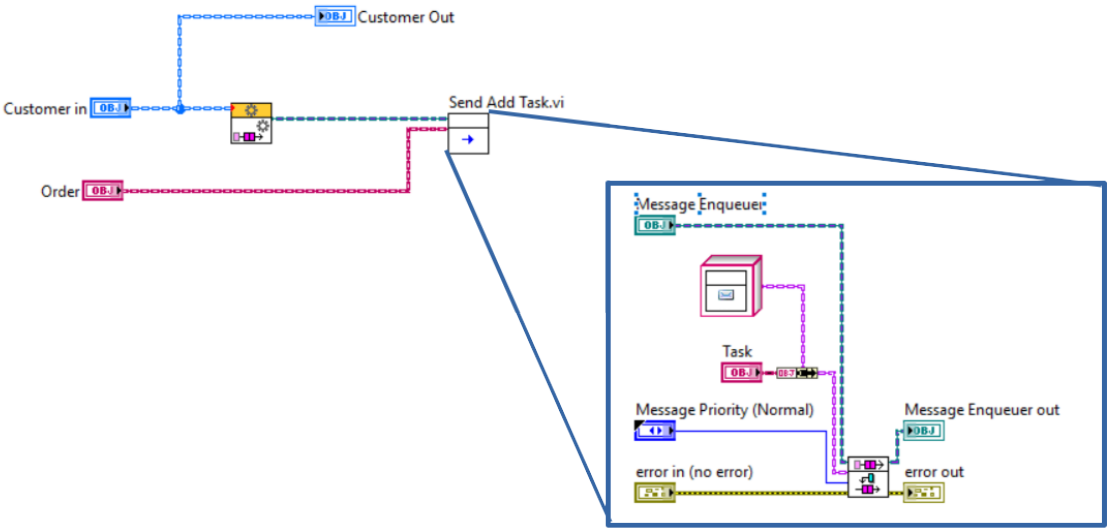
The main difference is that messages in Actor Framework are objects implementing the command pattern, while Triarc uses the string and variant message cluster used by the Queued Message Handler template.

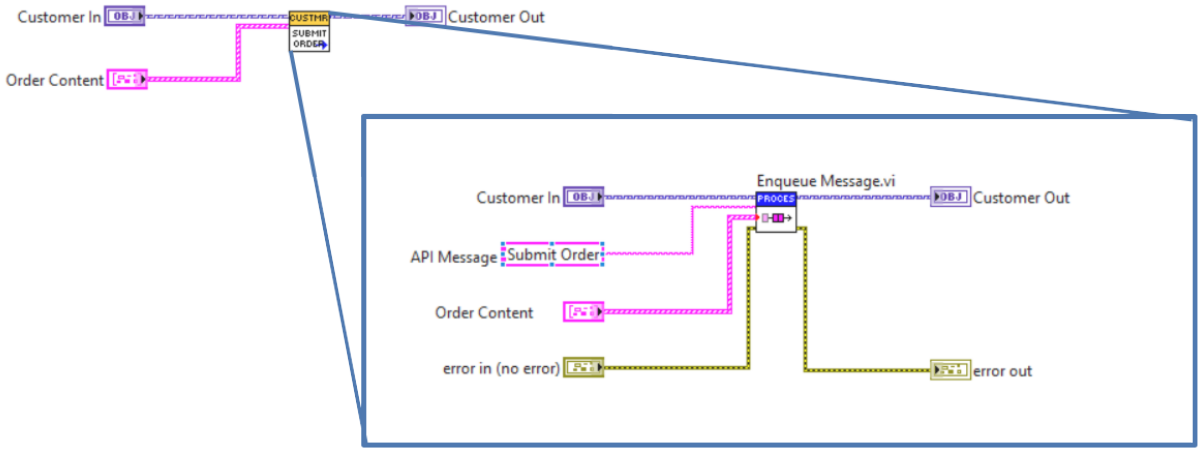
This section will go through some of the design decissions in the two frameworks and how they differ. For a better understanding, it is worth exploring the Coffee Shop example which ships with LabVIEW (called Actor Framework Fundamentals) and the counterpart implemented in Triarc.

19.2.1 Messaging

Actor Framework messages are sent by obtaining a reference to the enqueuer for the receiving actor and calling the Send method with the enqueuer as input. The enqueuer wire is a reference and cannot be unbundled to access the actor state outside the actor core process.

In Triarc messages are sent using the protected Enquque Message method. This method should be wrapped in an API VI which builds the message and the API is then called on the class wire to send the message. Conceptually the two are quite similar, as seen below.



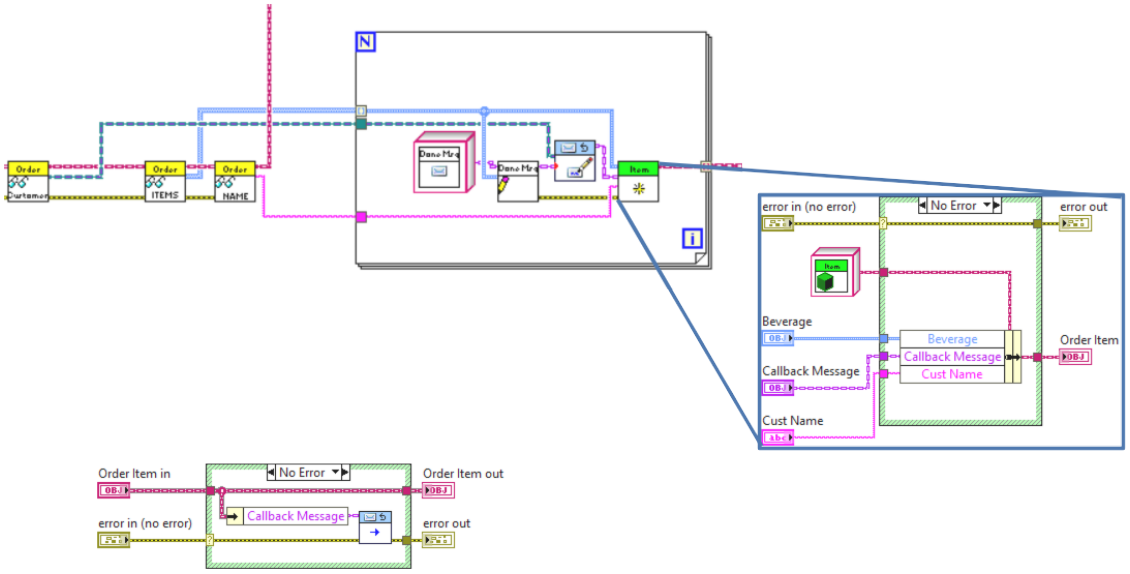


In actor framework the messages may be either abstract or coupled to a specific actor class hierarchy. An actor may allow its children to modify the bahvior of the Do method in a message by using dynamic dispatch methods which the children would then override.

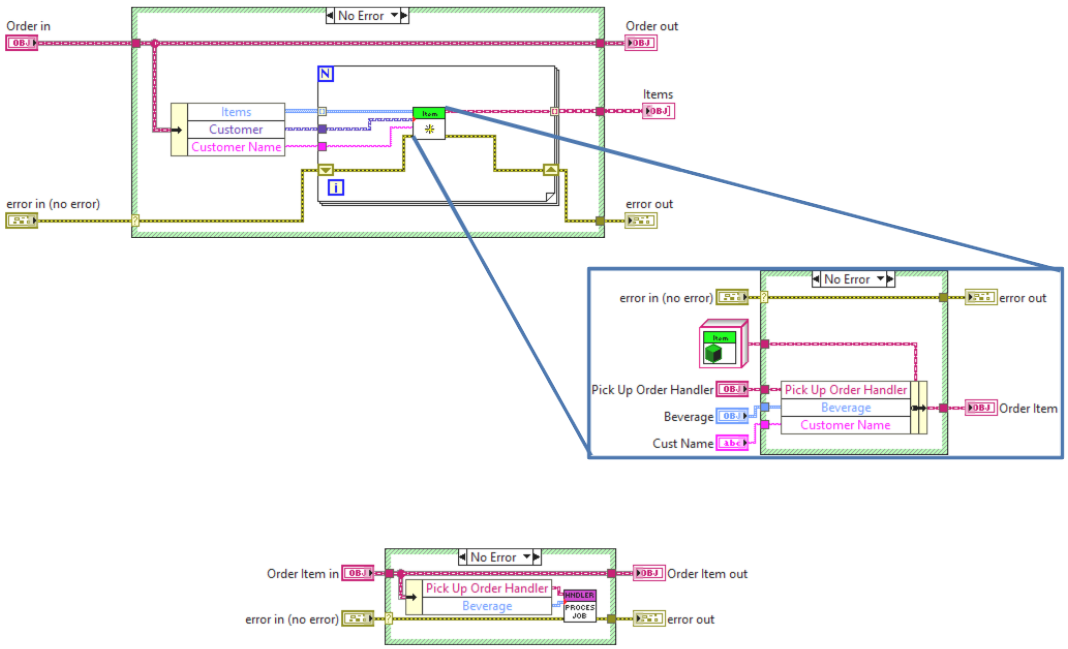
In Triarc the same concept may be implemented by adding cases in the Handle Messages.vi for the specific message in decending classes. An abstract message in Triarc is best implemented using an interface.

19.2.2 Callbacks

One nice advantage of using objects as messages is the possibility for defining callbacks. A message may be passed as a parameter to be called when a task is completed by another actor and the message may be forwareded between actors. An example of this can be found in the Coffee Shop example where the cahier builds a callback message which is sent with the order item to the barista. The barista fires the callback when the job is completed by sending the callback message.



To implement a callback in Triarc in a similar way, an interface must be created which the process handling the callback implements. The Callback Message in the data of the order item is replaced by an interface which defines the callback method. The callback message is sent by invoking the method defined by the interface.



Both methods works well and have the merit of decoupling the class making the callback from the class receiving it. As an interface cannot cary any data, the beverage input is given by the barista in Triarc, while the cashier is setting it in Actor Framework when constructing the message. In this example I think it makes more sense to have the barista set the order content to be picked up, but this is at the cost of slightly tighter coupling. To achieve the same decoupling as in the Actor Framework example and make the barista unaware of the return message content, the cashier could implement a more abstract interface used by the barista and the cashier could notify the customer using and asynchronous action.

19.2.3 Context awareness

In Actor Framework an actor may spawn nested actors, which may in turn spawn more actors. Nested actors may be stopped by the calling actor when on shutdown and nested actors may send messages to their caller by accessing the Callers Enqueuer.

In Triarc there is a class called Application which hosts processes or nested applications. Any process may access the application hosting it by using the Get Context method.

19.2.4 User Interfaces

In Actor Framework it is common to implement the user interface in the Actor Core.vi. Triarc introduces the concept of a View which implements a user interface component. The View separates the user interface from the process and provides some of the common boiler plate code.

19.2.5 Features

Actor Framework is more minimal than Triarc in the features available in the framework. The framework is responsible for handling messaging between actors, the actor lifecycles, error handling and debugging. Triarc provides more functioanlity, such as the View and Application classes, and the Helper Loop and Asynchrounous Action interfaces. Triarc also has configuration persistance features built in and support for logging and predefined automated unit tests.

Whether the additional functionality in Triarc is an advantage or not depends on how useful you find the features. Personally I have found that these are concepts needed for any project and should therefore be part of the framework. These provided concepts enforces a certain style and structure to the code.

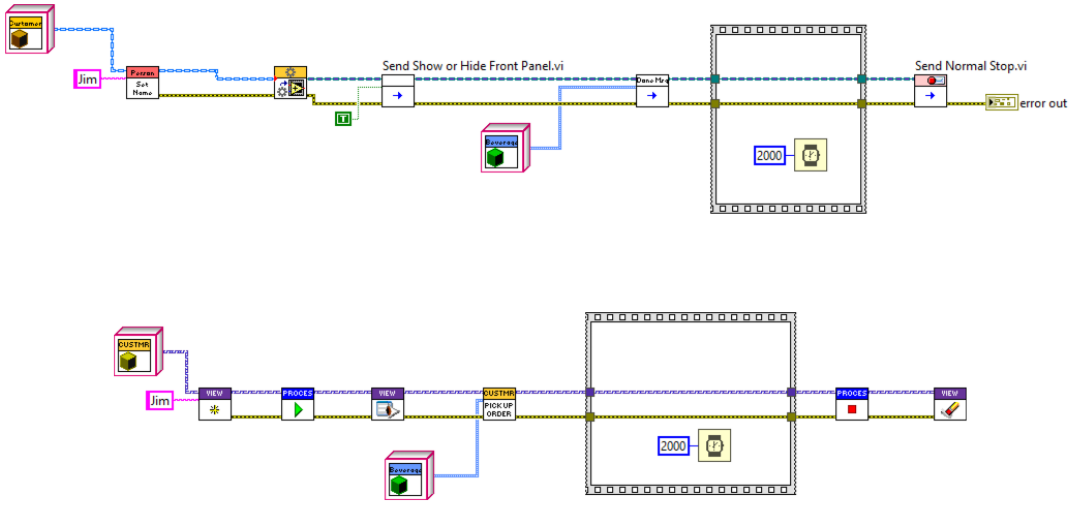
19.2.6 Ease of Use

This might be an oppinionated section, but one may argue that ease of use is an important aspect of any framework.

Actor framework is heavily object oriented, and by heavily I mean that every actor and message is an object. To determine what a specific actor does, you need to look at the actor class, then all messages for the specific actor and then all messages for all of its parent actors. If there are dynamic dispatch VIs used in the messages, you need to go back and find which one is actually called, which depends on the specific class. This might be a lot to diggest, even for experienced developers, and in my opinion it is unnecessarily complicated.

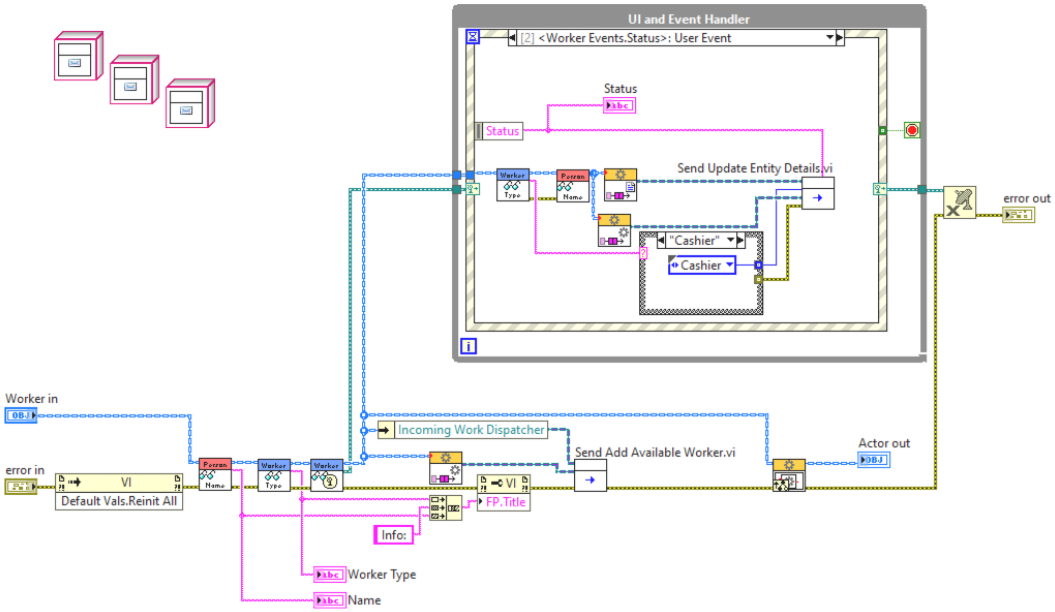
To send messages in Actor Framework you need to get access to the correct enqueuer and construct a message to be sent to the actor. This paradigm and the overall design of the interfaces in actor framework makes it look and feel very different from other well known designs, such as the queued message handler.

Triarc on the other hand is modelled after well known concepts so that it looks natural and intuitive even to an inexperienced programmer. Using a Triarc process looks very much like using an instrument driver, with well defined common methods for controlling the lifecycle and high level API-methods. Developing a Triarc process feels like developing a queueud message handler, but you do not need to care about the plumbing outside of the case structure. A comparison of the API for the same customer is shown bellow.

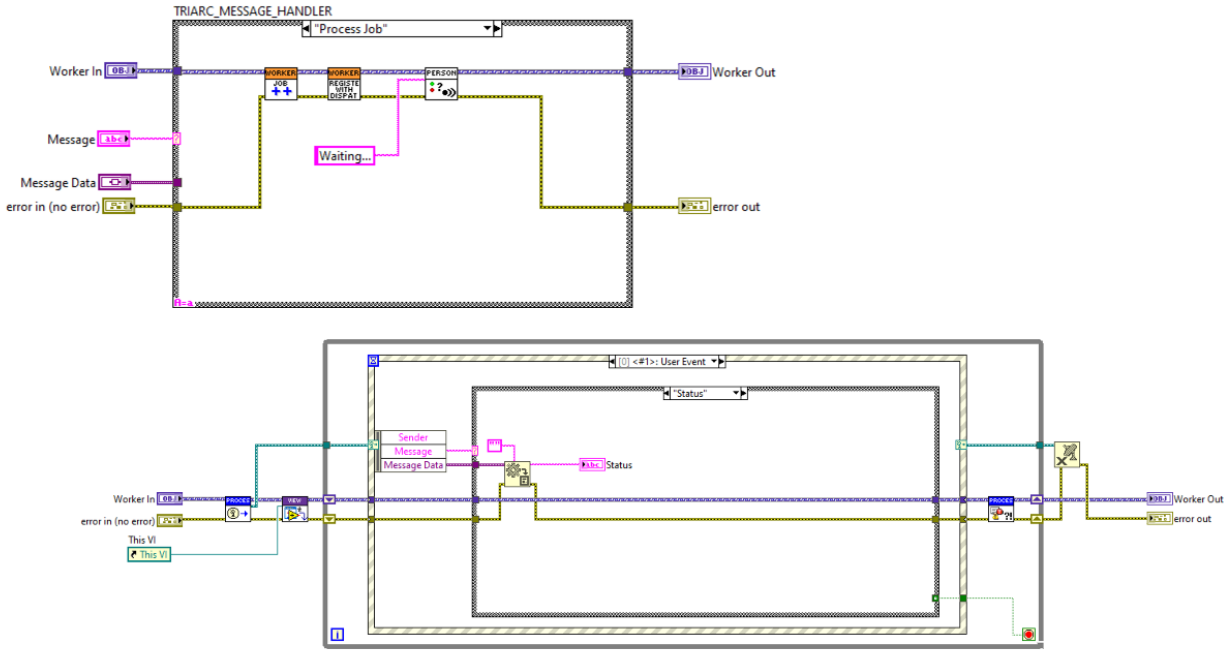


The main difference is not that the Triarc icons are more colorful, but the important distinction is that in Actor Framework VIs from the different message classes and the logic is spread out. In Triarc the methods defined for a class are all owned by the class or its parents.

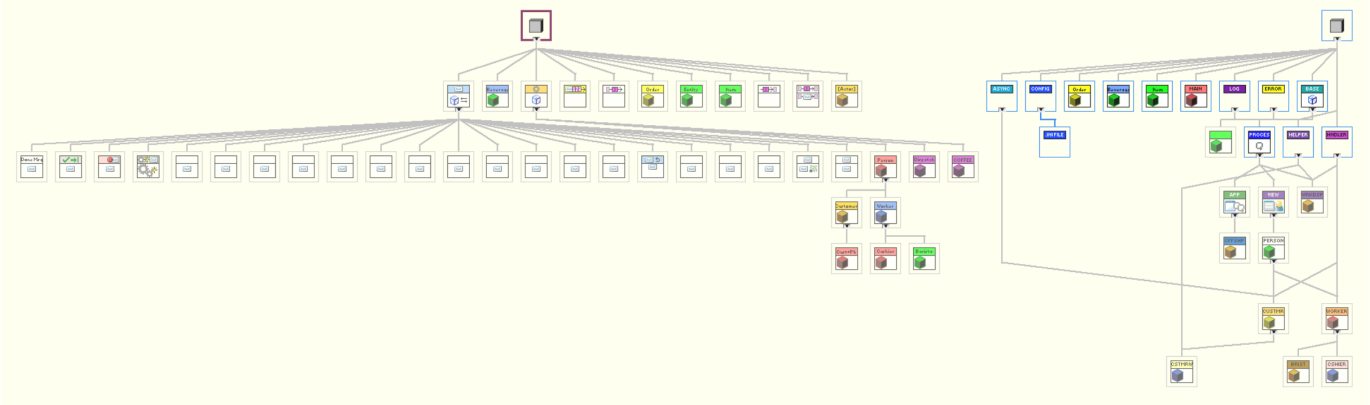
If we take a look at the implementation of the abstract(-ish) worker class in actor framework, the user interface is implemented in the actor core and the functionality is handled in the parent actor core and defined in the messages defined for the actor.



In Triarc the user interface and application logic is split up into separate VIs. The functionality is implemented in the Handle Messages VI and the user interface is in the View vi.



The amount of classes in a typical Actor Framework application tends to grow quickly as messages are added. As and example, the class hierarchy for the same application in Actor Framework on the left and Triarc on the right is shown below.



19.3 Conclusions

Actor framework and Triarc both implements different versions of the actor model. The main difference is how messages are represented as objects in actor framework and the functionality is implemented in these messages. To get a feel for the two frameworks, the coffee shop example is a very nice starting point.

20. Triarc and DQMH

LabVIEW has already existed for over 30 years and Triarc is not the first framework which has been proposed. Comparing frameworks is not a simple task as different frameworks have merits in different areas. There is simply no such thing as the best framework, it all comes down to who you ask. As of today, there are two frameworks which have achieved a somewhat broad user base. The first is NI Actor Framework is bundled with LabVIEW and is included in the installation of LabVIEW. The second is Delacor DQMH, which is an extenssion and improvement of the NI Queued Message Handler (QMH) template.

Disclaimer: I tend to be strongly opinionated and I am certainly biased, as I would not have created Triarc if I thought there was better alternatives out there for me. The purpose of this post is not to discourage the use of DQMH, it is not without reason it has become the most successful LabVIEW framework to date.

20.1 Brief introduction to DQMH

DQMH was invented by Fabiola De la Cueva and maintained by Delacor up until the DQMH consortium was founded. This framework is very approachable and builds on the QMH template shipped with LabVIEW and introduced in the LabVIEW training courses. The framework is built around modules, where each module is a library. Each module has a main.vi which, as the name suggests, is the main VI of the module. The main.vi contains two loops out of the box, where one handles events from the user interface or API calls, and the other loop is a message handling loop. The general idea is that events enter the module through the event loop of the main.vi and work is then enqueued to the message handling loop. A common design pattern is to implement helper loops in the main.vi which runs in parallel with the other loops. By heavily leveraging scripting, DQMH avoids the introduction of object orientation and greatly simplifies the creation of the events, which would otherwise be a quite tedious and error prone task. My personal favorite feature of DQMH is the Testers which comes with each module and facilitates manual testing of the module in isolation by making calls to the API.

DQMH is freely available, although not open source. A very big selling point of DQMH is the community and user base already committed to the framework. Some companies are qualified as DQMH trusted advisors and are certified to support customers using DQMH. The documentation is also extensive, including youtube videos, help files and comments in the framework code.

20.2 Comparing DQMH to Triarc

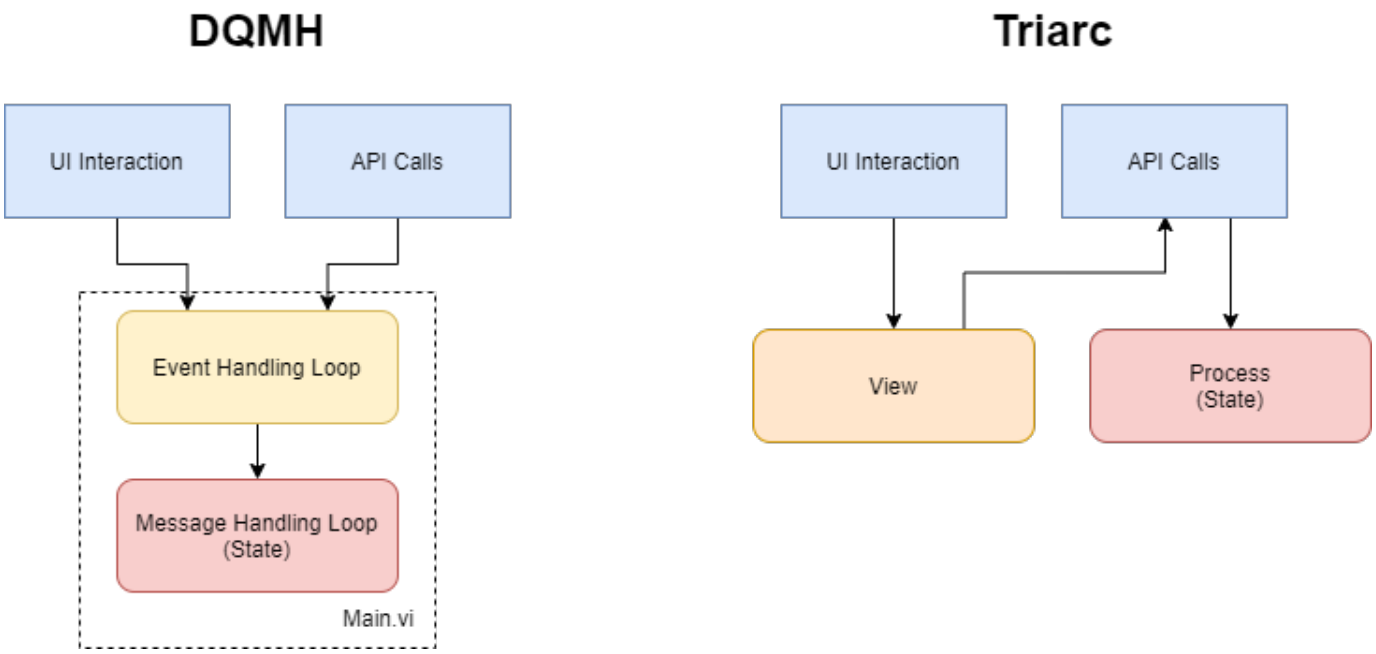
DQMH and Triarc are similar in many ways. A DQMH module corresponds to a Triarc Process, or to be more precise a Triarc View if the module UI is needed. DQMH goes pretty far to avoid object orientation, which may make the framework more approachable for very junior developers. Triarc is object oriented by design and each process is a subclass of the Triarc Process class.

This section will go through some of the design decissions in the two frameworks and how they differ. For a better understanding, it is worth exploring the Thermal Chamber example which ships with DQMH and the counterpart implemented in Triarc.

20.2.1 The Actor Model

Both Triarc and DQMH implements a version of the actor model, which is a well proven design pattern for concurrent systems. The main idea is that having an actor encapsulating a state and to only communicate with it by sending messages. In DQMH the actor role is taken by the DQMH module and the state is contained in the shift register of the message handling loop. Messages in DQMH are sent by generating events, which are captured by the event handling loop, and subsequently forwarded for processing in the message handling loop. This is very similar to the NI QMH template, but it has been generalized by allowing events to be generated externally using API calls.

In Triarc the process represents the actor in the Triarc framework and the state is maintained in the class private data of the process. Communication with the process is imlemented using API calls which enqueues messages to the process. If the process implements a user interface, it will inherit from the Triarc View class. The View class has a vi called View.vi which holds the user interface. Views may be composed in subpanels and the logic of each view is implemented in the processes of the independent views. The View.vi communicates with the process using regular API methods, or private methods for enqueueing to the process loop. The flow of information in the two paradigms are compared in the figure below.



20.2.2 Messaging

DQMH has two types of messages called requests and broadcasts. Requests are API calls typically sent into a known module from a caller and broadcasts are generated by the caller to respond to events without coupling to the listener of the broadcast.

In Triarc there is messages, requests and broadcasts. Messages are enqueued asynchronously, while requests are synchronous requsts blocking until a response is received. Broadcasts are very similar to the DQMH broadcasts, but they do not have the same type safety as the message data in a broadcast is a variant.

20.2.3 Hierarchical Structure

In DQMH there is no built in hierarchical structure for modules to adhere to. In good DQMH design however, modules are often ordered in a tree-like structure, but the responsibility for maintaining the order is on the developer. In Triarc the concept of an application and nested applications maintains the hierarchical structure and keeps them isolated from each other.

20.2.4 The Use of Global State

The DQMH module encapsulates framework references in a functional global variable (FGV) and this globally accessible variable is used to enqueue messages to the module. This is in fact global a global resource and using global resources requires careful consideration, especially when the application grows in size. As the references are accessible from anywhere through API calls, it is very easy to have modules call each other and the data flow may be broken up into many separate timelines. This might seem convenient, but it may also leat to the code being more obscure and API calls may be hidden within subVIs.

Modules calling API VIs on each other are explicitely coupled and if the design does not define the calling order for the modules, one might end up with very tight coupling between modules. This design makes it complicated to instantiate multiple instances of a module, which in the case of DQMH is solved by introducing a cloneable module which maintains an ID to filter messages by.

In Triarc the references used by the framework are kept within the class data of the process. This means that you are not able to interract with a process unless you have access to the wire of the process. This makes the API adhere to the principles of data flow even if the processes are running asynchronously.

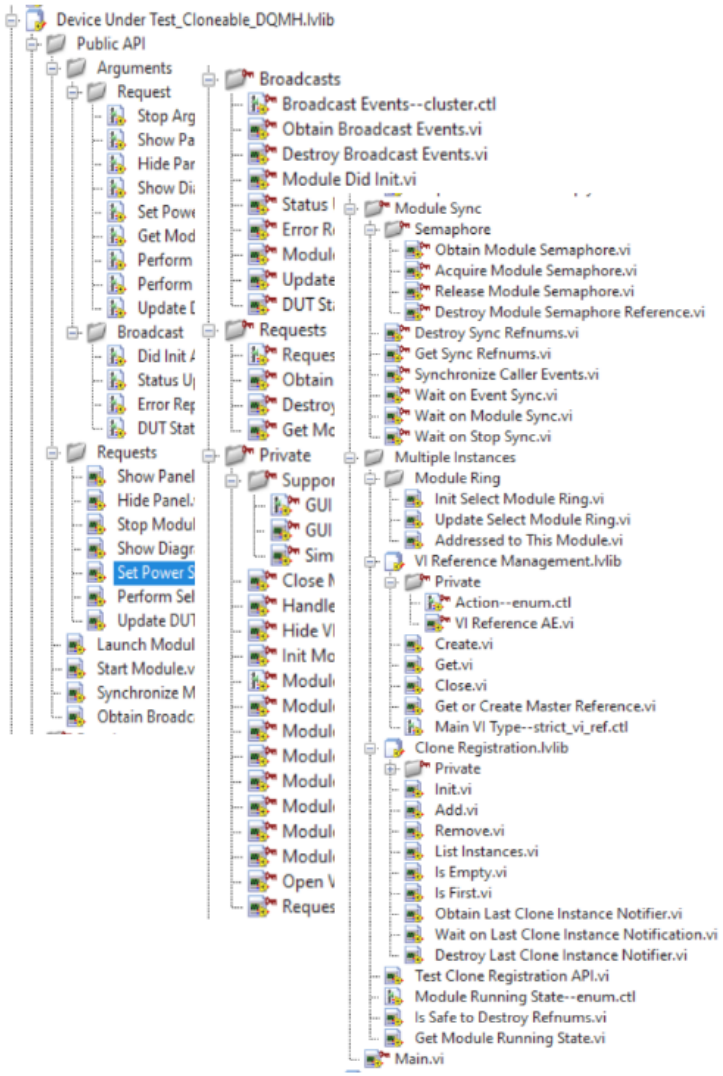
The Triarc framework is design with great care to prevent shared global state. There is not a single FGV, named queue, global variable, etc. in Triarc. If something needs to be globally accessible, which would often be the case with *e.g.* a log handler or error handler, the resource is injected on the top level VI, which is responsible for the lifecycle of the global resource. The lack of global state makes it very straight forward to instantiate multiple copies of the same Triarc Process.

20.2.5 Dryness

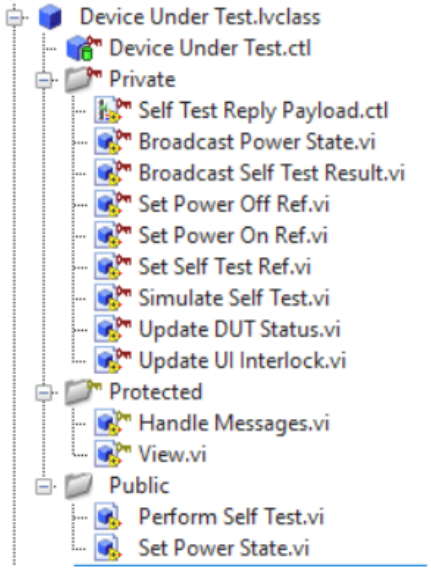
In software engineering there is a principle of not repeating the logic when writing code (the Dry principle). While one has to be pragmatic and make decissions which fits the design, having duplicate logic means having duplicate bugs and duplicate code to maintain.

In DQMH every module carries with it the full framework. One might argue that this is not an issue, as all the framework VIs are generated through scripting. In my opinion, the generated contet clutters the project and makes it difficult to get a quick overview of what the module API looks like. To illustrate this, let's have a look at the module for the device under test (DUT) in the thermal chamber template and compare it to the Triarc process implementing the same functionality.

DQMH



Triarc



Even if we were to lock the libraries to hide the private methods, there is still many times more stuff going on in the DQMH module. The reason we only need process specific VIs in the Triarc process is because all framework features are inherited from the main Triarc Process and View classes.

One advantage of bundling the framework with each module created is that modules may use different versions of the framework and still work within the same application. The flipside is that each module must be updated when new versions of the framework is released. In Triarc breaking changes will need to be solved by namespacing the framework within versioned libraries with an incremental naming convention.

20.2.6 Object Orientation

Whether object orientation is good or bad is beyond the scope of this post, but it does have a significant impact on the design. Triarc relies heavily on object oriented design, while DQMH only uses objects for certain subtasks.

One clear drawback of using a library over a class in LabVIEW, is the fact tha libraries are not first class citizens in LabVIEW whilde classes are. This means that a class may be wired on the block diagram just as any other data type, while a library cannot be used in the same way. The implications of this is that libraries cannot be composed into larger composite structures. Triarc processes are classes and therefore composeable.

By using object oriented design, powerful abstractions may be created and a lot of flexibility to define behavior at run-time is possible through subclassing. It also reduces the amount of code duplication as descending classes may use the parent class methods. A good example of this is the Show Panel.vi which is included in every DQMH module. In triarc this functionality is only defined in the View class and subclasses uses the VI from the View class to open therir panels.

20.2.7 Dependency Inversion

One of the main benefits of object oriented design is the separation of source code dependencies from run time dependencies. As DQMH is not object oriented, it is not possible to use abstractions to invert dependencies without wrapping DQMH modules in classes. This is a major limitation of the framework and it misses out on one of the core ideas of object oriented design.

20.2.8 Type Safety

In DQMH each event has an associated data type defined by a typedefined cluster. The events are generated through scripting, so the tedious task of setting this up is handled by the framework. The outcome is that each event case has statically defined types which will give the correct types at edit time. This can be pretty useful to ensure bad type casts.

In Triarc the data is flattened to a variant in each API VI and sent to the process as a variant. This puts the responsibility on the developer to cast the variant to the expected data type. In practice this is not too painful, as the pipeline the data goes through is well defined. It is more complicated for broadcasts, as each listener needs to know what data type to expect.

The dynamic typing will require more testing and preferably automated testing to prevent regression issues. On the other hand, having the correct type is not equivalent with correct behavior and we should probably be doing the testing regardless.