User Guide

ioflo version 0.9.03

Document Revision 0.9.03.0

Samuel M. Smith PhD

242 East 600 North
Lindon
Utah 84042-1662
T: USA
F: 801.766.3529
E: sam@ioflo.com
W: www.ioflo.com

2014/01/29

Table of Contents

1. Introduction

1.1. Core Architecture

Floscript is the the configuration language for the ioflo runtime engine. floscript supports the configuration and scheduling of reconfigurable component software modules that interface with a publish/subscribe shared data store. These software components allow convenient expression of control and planning algorithms as well as transparent monitoring, logging, and replay through the distributed publish/subscribe shared data store. The dependency reduction arising from modular components interfacing to a common store significantly reduces apparent complexity. The architecture infra-structure is unique in that it seamlessly unifies the scalable distributed data flow or port based component paradigm with the power and expressiveness of hierarchical discrete event systems. The architecture enables the formation of almost any semi-autonomous command and control system organization in a highly convenient manner.

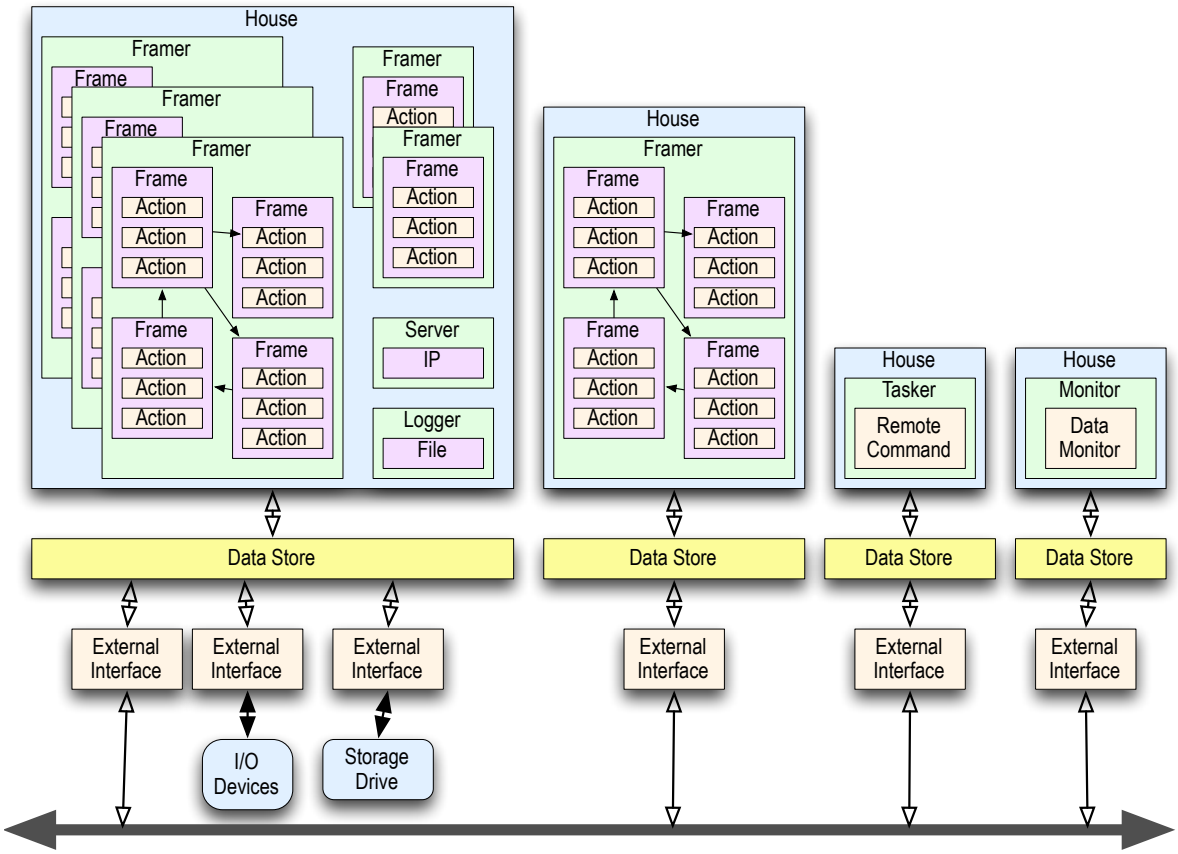

Fig.1.1: Core Component Based Pub/Sub Architecture

2. Hierarchical "State Machine"

In addition to information exchange an autonomy architecture needs a way of configuring, organizing, scheduling, sequencing, and evaluating sensing, command, and control components.

Each component action, agent, controller, etc in an autonomous command and control system using conventional computation equipment is at some level a form of state machine. We use the term "state machine" loosely here. By Hierarchical State Machine we mean a means of describing the evolution of a system through various states where those states are composed in a hierarchical manner. This is the sense used by StateCharts and UML (Unified Modeling Language). "State machines" provide a very convenient formalism for modeling component behavior. Moreover, autonomous control usually involves some sort of mission plan, that is expressed as a sequence of stages or activities, in other words a form of state machine as well. Various architectures use different levels of granularity to express this stages along a spectrum.

Whether explicit or implicit, any autonomy software architecture running on a computer platform is a hierarchy of state machines. Moreover, human cognition is limited in how many distinct pieces of information can be thought of at one time. Thus it is difficult for a human to perceive simultaneously all the constituents of a complex system. Hierarchical composition/decomposition is one way, if not the only way, for humans to design and manage complex systems. More complex systems are built up from simpler subsystems where the mutual dependencies at each level are simple enough to be managed. Indeed it is the lack of a sufficiently general and flexible mechanism for hierarchical composition of the software components that eventually limits many autonomy architectures thus creating an apparent complexity barrier.

Since hierarchical composition is the "natural" way for humans to design complex systems, a hierarchical state machine would appear to be the most appealing approach. Indeed, ioflo has at its core a hierarchical state machine modeling paradigm.

3. Hierarchical Action Framework

Although UML (Unified Modeling Language) and State Charts each provide a rich environment for designing, modeling, and even executing hierarchical state machines, they are both relatively heavy weight implementations. What we wanted was support for HSMs in a light weight easily embeddable implementation. We wanted a simpler syntax that implements the core features needed for intelligent autonomy without unnecessary cruft. The main purpose of the hierarchical state machine in an intelligent autonomy application is to sequence, schedule, or queue actions to be performed by the vehicle. After doing some use case analysis, it became apparent that all the generic features of HSMs as implemented by UML or State Charts are not needed for AUV control. A more efficient implementation was possible that focused on the main task of scheduling "actions". By careful combination of a simpler but more modular set of capabilities all the use cases could be supported with ioflo, especially when the emphasis is on scheduling actions and behaviors. One of the key innovations of the ioflo is the use of generators in Python that enable weightless or micro threads (cooperative co-routines).

3.1. ioflo Coding Metaphor

One way to make source code more understandable and easier to develop and maintain is to use a metaphor or allegory for the naming and interrelationships of the software components. A powerful metaphor can help make the function of various elements and components intuitively obvious without introducing confusion with other applications or implementations. Because state machines are ubiquitous they bring a certain baggage with them. In addition, state machines for intelligent autonomy applications need some unique features. Consequently, we decided to use a Metaphor for ioflo that is uncommon for state machines in general but has an internal logic that is consistent with

the concept of an HSM. For a given metaphor, the English language may have many synonymous terms for elements of the metaphor. For the sake of convenience and economy of expression, shorter terms that fit the metaphor are better than longer terms even if less common. Uncommon terms or terms not used before in a similar application (autonomous control software) also have the benefit of having less baggage.

In common usage a *frame* is a structure for admitting or enclosing something. In the ioflo metaphor, a frame encloses a set of actions to be performed by the vehicle.

One of the distinguishing features of a state machine is that it can change state. The change in state is usually called a transition. So actions executed by a state machine can be split into two types, 1) actions that are meant to be performed while in a given state, that is, *same state* actions, and 2) actions that cause the state machine to transition or change to a different state, that is, *transition* actions. In a hierarchical state machine, the "state" at any point of time is composed of a union of sub-states at different levels in the hierarchy. In ioflo this union of sub-states is represented as an ordered list of frames, we call an *outline*.

An ioflo framework can be thought of as a hierarchical action framework that consists of a multiple levels of frames stacked on top of each other with a defined relationship between levels in a tree-root like structure. Tracing a path from the trunk down to an end root produces a list of frames, we call an *outline*. The current state is the ordered union of all the frames in a given outline. The current state's outline is the *Active* outline. It is just a list of frames that goes from the top level down to the bottom level. It metaphorically traces an outline through the hierarchy that determines a given state.

A familiar software hierarchy exists in object oriented systems via inheritance, in which sub classes shadow actions of super classes such that if a sub class action exists the super class action is not executed unless explicitly called by the sub class. Thus with object orientation, there is really only one action not a hierarchy of actions. This type of hierarchy is not appropriate for frame action hierarchy in ioflo. Consequently we could not leverage OOP inheritance directly for the ioflo hierarchy. Instead, in ioflo. each level is meant to be executed. The execution order is from the top down to the bottom, that is, higher level frames are always executed first and are not shadowed, but may be overridden.

This has three beneficial characteristics.

1) Higher level frames can be used for common actions to greater simplify the complexity of the ioflo and provide greater economy of expression.

2) Transitions higher in the lineage have higher priority, this allows higher frames to be used for safety checks that result in transitions to recovery states when a problem occurs. Lower level frames can focus on the apparent mission goals without being complicated with safety checks.

3) When a transition occurs from a higher level frame, the lower level frame actions are not executed, thus allowing higher level transitions to override lower level ones. When no transition occurs from a higher level frame, then lower level frame actions can override (undo but not shadow)higher level frame actions. This allows the lower level frames to take priority in actions but not transitions. This is good because in the case that events indicate that the machine should stay in a particular state, that is, do the actions for that particular state, then the lower level frames will have priority to express actions with higher specificity for the current state than the higher level frames. And when events indicate that the current state is not appropriate then higher level frames will have priority to make transitions to states better suited to the current situation.

Indeed it is the dichotomy between priorities for *same state* actions and *transition* actions that makes a hierarchical action framework so powerful.

The disadvantage of a ioflo is that if a lower level frame action overrides an action in a higher level frame there may be wasted computation, although a smart loader/parser/scheduler could ameliorate this by determining if an action in a high level frame will be overridden by an action in a lower level frame and then not perform the higher level version.

Each ioflo is executed, evaluated, or run by a manager. We call this manager a Framer. In other words a ioflo or Framework is run by a Framer.

Because each autonomous system or vehicle may need to have multiple frameworks executing in parallel, some object is needed to keep track of these frameworks. In keeping with the metaphor, this object is called a *House*. A *House* holds a list of all of its frameworks as well as a reference to the data store shared by all the frameworks in the *House*.

3.1.1. Hierarchies of ioflo HAFs

Because each framework consists of multiple levels of frames, it is in this sense hierarchical. There is another sense, however, in which a hierarchy could be imposed and that is to have one Framer control another Framer. This is a hierarchy of Framers. In this sense we then would have a hierarchy of hierarchies. Both senses are implemented in the ioflo architecture. A Framer runs a hierarchy of frames which in turn could each contain or control Framer(s) for other ioflo HAFs. The advantage of this division is that a common sequence of actions that might be used more than once during a mission can be expressed as a ioflo HAF. This ioflo HAF can then executed by different frames in the overarching mission framework.

Yet another level of state machine hierarchy is supported by ioflo. The individual actions or behaviors executed in each Frame can also be state machines in their own right. But these internal state machines are not exposed to the ioflo HAF.

3.1.2. Naming Conventions

Our convention for naming a software list or dictionary of objects is to use either the plural form of the noun or a collective noun. For example, a list of frames could be called *actives* for the active frames or *outline* for a list of frames in the current state.

3.2. Architecture Component Descriptions

3.2.1. Development Environment

Ioflo is implemented with Python, which is an extensible, open source, fourth generation development language with support for advanced computing techniques including functional programming, OOP, and weightless-threads. Python can be extended using C for high speed operation without sacrificing its expressive power for rapid development. Python is supported on all major operating systems and is embeddable. Python is used extensively in the science community as well as by industry leaders such as Google. Because Python is a dynamic byte compiled language with a very powerful but simple syntax, it by itself significantly reduces apparent complexity and perceived risk.

Python supports "generators" that can be used to implement weightless cooperative micro threads. The term weightless comes from the fact that the overhead associated with the generator is negligible,

no more than a function call (no process environment swapping). This allows tens of thousands of threads to run on a single processor. This is a revolution in distributed computing technology. Indeed, one of the most significant innovations of ioflo is the pervasive use of weightless threads to reduce apparent complexity.

One of the difficulties in implementing high levels of intelligent autonomy is dealing with the dichotomy between the real world which is inherently concurrent and the computational world which is inherently serial. The conceptually simple way of modeling a real world situation is to run each element of the real world as an independent concurrent process.

The concept of threading was introduced to enable serial computers to simulate real concurrent actions. Conceptually, each thread believes it is operating concurrently with other threads even though physically the threads are time sliced. When running, each thread is given an environment with processor resources such as a call stack, registers, etc. The time and memory overhead associated with switching environments between threads as well as managing access to shared resources, precludes large number of threads from running on a single processor. Large in this case means hundreds or at most thousands.

Thus to model or simulate real world systems using the natural model of concurrency and conventional threads has until now required extensive computation resources. Weightless threads remove this computational barrier. The key idea of a weightless thread is that the granularity of a recurrent "process" is made very small, down to the individual function level. The threading becomes part of the language instead of the operating system. The following code snippet illustrates how simple it is to implement basic weightless threads in Python.

```
#Scheduler
def scheduler(threads):
   while 1:
      for thread in threads:
         status = thread.send(command)
         #scheduler housekeeping here

#Generators as Micro-thread
def thread1():
   while 1:
      command = yield(status)
      #do thread's task here

def thread2():
   while 1:
      command = yield(status)
      #do thread's task here

#Execute
threads.append(thread1())
threads.append(thread2())

scheduler(threads)
```

Weightless threads have recently been adopted by segments of the computer gaming industry for character AI and also by segments of the discrete event simulation community. With weightless threads, the nature of autonomous control algorithms and architectures is fundamentally changed. Tens of thousands of weightless-threads enable on a single processor the richest possible inclusion and management of real world constraints into the computation process. Each mission objective,

constraint, sensor, controller or actuator gets its own concurrent "agent". Each agent runs in its own weightless thread.

Python with weightless threads uniquely forms a "sweet spot" for implementing the intelligence needed for high fidelity software agents. Weightless threads in a high level language achieve computational efficiency in a programmer friendly way. Power of expression is the sine qua non of this approach. Practical attainability comes from a judicious combination of computational and programmer resources. The fundamental technical advantage of this approach is an exponential increase in computational efficiency without a corresponding exponential decrease in programmer efficiency.

In ioflo the various tasks are all implemented as weightless threads.

3.2.2. State

A state is a hierarchical combination of frames, with one frame from each level of the hierarchy. A state is represented by an *outline* of frames (see below)

3.2.3. Frame

A *Frame* is a collection of actions. Each Frame stores different types of actions to be performed in different contexts while in a given state.

Each Frame may be attached to one and only one Frame in the level above if the level above exists and may be attached to at least one Frame in the level below if the level below exists. A Frame may be attached to multiple frames in the level below but only one of these is the primary.

The Frame in the level above is called the *Over* Frame. Each Frame has a link to its Over Frame or None if the Frame itself is on the top level. A *Top* Frame is defined as a frame whose Over is None.

The primary Frame in the level below is called the *Under* Frame. The collection of frames in the level below is called the *Unders*. Each Frame has a link to its Primary Under Frame called Under and a list of all the under frames, including the primary, called Unders. Under may be None if the Frame itself is on the bottom level. A *Bottom* frame is defined as a frame whose Under is None. The Under frame defaults to the first frame attached underneath but may be changed to another frame attached later, that is, any of its Unders.

3.2.4. Outline

An *Outline* is an ordered list of frames that traces a path from the top to bottom level through the frame hierarchy. Specifically, an outline is a list of frames that traces the Over Frames starting at a given Bottom frame all the way to a Top frame. An Outline composes a state. Outlines in the framework may be of different lengths.

Because each Frame has a single primary Under frame, an Outline can be uniquely defined for any Frame not just Frames on the bottom by recursively tracing the Over links from that frame up to the top and the Under (primary) links from that frame down to the bottom. Thus for each Frame there is one and only one Outline derived by beginning a trace at that frame. This is the Frame's outline. For a given outline, a *Beginning* frame is a frame who's outline is the given outline, this is, if a trace is begun at a *Beginning* frame it will result in the associated outline. In other words, although a frame may be a member of multiple outlines, it is a Beginning frame of only one outline. A given outline may also have more than one Beginning frame.

3.2.5. Transition

In addition to Over, Under and Unders links, a given Frame may also have transition actions that link to other Frames. In a transition the frame where the transition begins is called the Near Frame and the frame where the transition ends up is called the Far Frame. Another way of referring to the Far Frame is that it is the Target of a transition. Because each frame is uniquely associated as a beginning frame of only one outline, without loss of generality, a state transition between corresponding outlines may be specified by a transition between two frames, one from each outline. The associated outlines can either be pre-computed or derived from the over and under links of the associated frames.

3.2.6. Framework

A Framework is more formally defined as the the complete set of frames connected by Over, Under, Unders, and Far links that are managed by a given Framer. A framework may have multiple Top frames. In other words the frames in a framework do not all have to descend from a single Top frame.

3.2.7. Active Outline

At any given instant of time only one outline from a framework is *Active*.

Uniquely associated with each framework is a Framer that runs or operates the framework. In other words, each framework belongs to only one framer and each framer runs only one framework. A Framer keeps a reference to the Active outline. Usually, the Active outline is the associated outline defined by the Active frame for the Framework and its associated Framer. The Active outline is a list of frames called Actives.

Framers are implemented as objects that have a Python generator for evaluating the framework

The actual objects for Frames and Framers contain many attributes and methods not described here. The main purpose of this description is to define the architecture of Frames, Framers, and Links that constitute a FrameWork. The specific objects are defined later in more detail.

3.2.8. Shared Data Store

Information is made observable and can be shared between frames and framers using a shared data store or *Store*. An item in the *Store* is a *Share*. Each Share may be a single datum or a collection of data. In addition to its Data each Share has several other attributes, these are:

Store = link to its associated Store

Name = unique ascii string that is external identifier for the share in its Store

Owner = Framer that owns the Share , that is, holds the action that writes to the Share

Stamp = time when Share data was last updated.

3.2.9. Tasker

Active Framers are executed by a Tasker. The Tasker automatically and periodically runs the python generator associated with the Framer. Each Framer has an execution status. These are as follows: Readied, Stopped, Started, Running, and Aborted. The Tasker sends controls to the generators associated with each framer to change the execution status as appropriate. These controls are as follows: Ready, Stop, Start, Run, Abort. The combination of the status and control allow the Tasker to

manage the execution of a given Framer. An Active Framer is one that is automatically started and run by the Tasker. Once Started it automatically receives periodic Run controls from the Tasker. An Inactive Framer is one that is Stopped waiting upon some event for the tasker to Start it and then Run it. Once it has been started it is reactivated and the Tasker will automatically continue to Run it. There are two other types of Framers, namely, Auxiliary and Slave. These are not run directly by the Tasker.

3.2.10. Auxiliaries and Mains

An *Auxiliary* Framer (Aux for short) is a Framer that is run by another Framer in the following way. Each Aux Framer is attached to a Frame in its controlling Framer. This Frame is called the Main Frame . Each Main Frame may have zero, one, or more Aux Framers. The Framer that runs the Main Frame for a given *Aux* is the Main Framer. The actions in an Aux Framer are sequenced automatically whenever its Main Frame is run. The generator based Framer controls and Framer status are not used. Essentially, the actions in the Aux extend the Main Frame with more actions. This allows common sub sequences of actions to be coded as an Auxiliary and then reused. Auxiliary Framers serve the same purposes in ioflo as subroutines do in a conventional stack based software program. The difference is that Auxiliaries fit naturally within the ioflo paradigm and preserve the traceability and observability characteristics. Indeed one of the unique contributions of ioflo is its convenient and novel implementation of Auxiliary Framers. This was not an easy feature to implement.

Auxiliaries provide a hierarchy of hierarchical action frameworks. This hierarchy of Auxiliary Framers, however, is constrained, to avoid concurrency problems, such that, only one Main Frame for a given Aux is active at a time. The purpose of the Auxiliary Framer hierarchy is to allow the reuse of frameworks or more appropriately framework fragments without having to create multiple mostly redundant copies. Each Main Frame can provide a unique data store context for the execution of the Aux Framer.

3.2.11. Slave Framers

In some applications it is necessary to have very explicit control over the sequencing of frames in a framework. The sequencing may need to be paused or only advanced upon certain events. This means that one framework is used to manually control the sequencing of another framework. This is supported in ioflo through the use of *Slave* Tasks or *Slave* Framers. A Slave Framer is only sequenced upon an explicit control action from a Frame in another Framer. These control actions are equivalents of the control actions the Skedder uses to control task-able Framers. The Frame/Framer holding the command actions is called the Master Frame/Framer. This is in contra-distinction to Auxiliary Framers which are executed automatically whenever the Main Frame is active. Slave Framers follow the more conventional paradigm for a hierarchy of hierarchical state machines. Because Slave Framers are not scheduled directly by the Skedder, the only concurrency issue is to make sure that only one Master for a given Slave is active at a time.

3.3. Actions

3.3.1. Action Execution Contexts

Each Frame holds *Actions* that are to be executed when a Frame is in an active Outline. *Action* is a generic term that actually is implemented by several python objects working together. The Actions are of different types. All Actions essentially consist of either directly changing something in the Store or running software that changes the Store.

Actions are executed in several contexts. These execution contexts have a shorthand name. Some contexts are only executed upon satisfaction of associated *conditions*. In common state machine parlance, these *conditions* are called *guards*. We simply refer to them as *conditions*. In keeping with our paradigm naming convention we have invented some descriptive shorthand names for the contexts. The context names and descriptions follow:

1) *benter* -A condition that must evaluated to True Before Entry into the associated Frame is allowed.
2) *enter* -An action that is executed at frame Entry, that is the first time a frame is executed as part of a new active outline.
3) *recur* -An action that is executed at recurrence of the Frame, that is, whenever the frame is iterated without a change in the outline.
4) *exit* - An action that is executed at frame Exit, that is the last time a frame is executed as part of an active outline
5) *precur* - An action that is executed Prior to Recurrence such as transition actions or setup actions for transition actions. If a transition condition is True then the outline is changed before any other frame actions are executed.
6) *renter* - An action that is executed upon Re-Entry into a Frame, that is, whenever the associated frame is a common member of the active outline both before and after a transition and the frame is above the target in the outline.
7) *rexit* - An action that is executed upon Re-Exit from a Frame, that is, whenever the associated frame is a common member of the active outline both before and after a transition and the frame is above the target in the outline. The difference between rexit and renter contexts is the execution order of the associated Frames.
8) *aux* - This is a hybrid context. It is not a single context but represents how the actions from auxiliaries are executed in the main frame execution contexts.

We suggest first understanding the first five contexts as these are the main use cases. Then once these are grasped, it will be easier to appreciate the distinctions offered by the last 3. Each type of Action has a *native* context that is used by default when an explicit context is not specified. Some actions may only be assigned to their native context, while other Actions allow their native context to be overridden. These contexts are an integral part of understanding how the ioflo works.

Within each Frame is a list holding the associated actions in each context. The list names and contexts are as follows.

beacts = *benter* actions (before entry).

enacts = *enter* actions.

reacts = *recur* actions (recurrence)

exacts = *exit* actions

preacts = *precur* actions (prior to recurrence)

renacts = *renter* actions (re-enter)

rexacts = *rexit* actions (re-exit)

auxes = auxiliary framers

The purpose of benter actions (beacts) are to provide checks or conditions (guards) that prevent entry into the frame unless the conditions are met. These conditions might be resources, or a certain state of the vehicle, that the other actions in the frame are dependent on. When an outline of frames is a candidate for entry, the entry conditions (beacts) are evaluated from the top down, that is, the beacts of the top most frame are evaluated first and then if True the next frame's beacts are evaluated and so forth.

The purpose of enter actions (enacts) are to configure or set up the rest of the activity within the frame. The purpose of exit actions (exacts) are to clean up or restore a configuration before leaving a frame. Consequently, enter and exit actions are executed in nested order so that configuration and clean up occur symmetrically at each level of the frame hierarchy. When an outline of frames is entered, the frames are entered from the topmost down, that is the enacts in the top most frame evaluated first and so forth. When an outline of frames is to be exited, the frames are exited from the bottom up, that is, the exacts in the bottom most frame are executed first and so forth. This nested symmetry is shown in the following diagram.

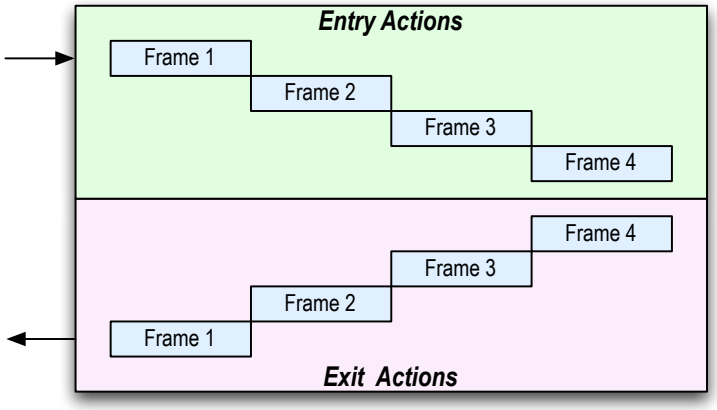

Fig.3.1: Nested Execution Order of Entry and Exit Actions

The purpose of recur actions (reacts) are to repeatedly execute something every time the associated Framer iterates. An example react might be a controller or filter that operates on a stream of incoming data. When an outline of frames are to be recurred, they are executed in top down order, that is, the reacts in the top most frame are executed first and so forth.

The purpose of precur actions (preacts) are to perform transitions between outlines in the ioflo. Typically a preact consists of a transition condition and a transition target frame. If the condition is met then the target frame's outline is to become the new active outline. A preact could also be some action that just performs a computation to setup the transition condition of a later transition type preact. The purpose of using a preact to set up the transition conditions is that it occurs on the same time step just before the transition conditions are evaluated. The setup usually means computing any necessary Data Store values used by the transition pre-condition. If such timely setup is not needed then the setup actions could be performed as a react. Finally a preact could perform a conditional auxiliary. These will be described later. When an outline of frames is to be precurred, they are executed in top down order, that, the preacts in the top most frame are executed first and so forth. If a successful transition is found at a higher level then the lower level preacts will not be evaluated on

that iteration of the Framer. Lower level preacts can only override (undo but not shadow) higher level preacts, if there are no successful higher level transitions.

The purpose of renter actions (renacts) and rexit actions (rexacts) are for special circumstances where a given frame wants to do something when a transition occurs between lower level frames in the active outline but the given frame is not explicitly exited or entered. An example might be incrementing a counter.

The auxes are not actions but references to auxiliary frames whose actions will be automatically executed in contexts related to the actions types.

3.3.2. Action Execution Order

A Framer is designed to run repeatedly at a given rate or with a given repeat interval or period. The repetition rate for the Framer is usually selected to be the highest sample rate needed by the reacts in the framework.

We will now describe more specifically the execution order for a Framer when there are no auxiliaries. The case with auxiliaries is described later.

The first time a Framer starts running it must be given a beginning frame outline. This outline becomes the active outline for the Framer and can be generated by specifying the beginning active frame.

The Framer checks the *beacts* of all the Frames in the active outline from the top down. If these are not all True, the Framer won't start.

If they are all True then the Framer evaluates the *enacts* for all the Frames in the active outline from the top down.

Next, the Framer executes all the *reacts* for all Frames in the outline from the top down.

On this first run the Framer does not evaluate any *preacts* which may include transitions. Transitions are special because they change the order of execution of actions. Thus each outline once entered is guaranteed to run all *enacts* and *reacts* at least once.

On each subsequent iteration (after the first one), The Framer evaluates the *preacts* first. These are executed for each frame in the outline from the top down. As mentioned previously, there are three types of actions associated with *preacts*. The first type are transitions, or *transiters*. These have a transition condition and a transition target. The transition target defines a new outline that becomes the new active outline should the transition succeed. The detailed steps to evaluating a transition are explained separately below. Transitions must return False if not successful. The second type of preact is an action that sets up a transition condition for a subsequent transition action. These setup actions must always return False. The third type of preact, is a conditional auxiliary which uses a *suspender* . A conditional auxiliary is similar to a transition, in that it has a condition and a target but the target is an auxiliary framer not another frame. The details of conditional auxiliary evaluation are explained in more detail further below. A conditional auxiliary action must return False if not successful.

Evaluating a transition is multi-step process that all occurs within the same iteration of the Framer. First the condition for transition is checked. If the condition evaluates to True then the Framer computes the sets of frames that will be explicitly exited, and explicitly entered, and implicitly re-exited and re-entered as a result of performing the transition .

In the simplest situation, the targeted frame is not in the current active outline, although higher level frames may be. In this situation, the explicitly exited frames are those frames that are in of the current active outline but not in the active outline specified by the target frame of the transition and the explicitly entered frames are those frames that are in the new active outline but were not part of the current active outline. The notion is that higher level frames that are common to both the current and targeted outline have already been setup so their enter actions should not need to be re-executed nor should their exit actions be executed since they are still part of the active outline. Frames that are common to the current active and targeted outline are not explicitly exited or entered. These are the frames that will be implicitly re-exited and re-entered.

For example , if the current active outline is represented by the list of frames, `[a, b, c, d, e, f, g]` and the target of the transition is frame `h` with outline `[a, b, c, d, h, i]`, then a transition from any of the frames in `[a, b, c, d, e, f, g]` to frame `h` would produce explicit exit frames = `[e, f, g]`, explicit enter frames = `[h, i]` and implicit "re-exit"/"re-enter" frames = `[a, b, c, d]`.

We discovered some use cases where one might wish to execute some actions only when a frame is implicitly exited and entered as part of a transition, i.e. is a common frame to both the near and far outlines. One of these is to increment a counter. The *rexit* and *renter* contexts support these use cases.

A more complicated situation is when the target frame is a frame in the currently active outline. One could deal with this situation in three different approaches:

1) Treat is as an error and disallow it.

2) Make it innocuous by finding the first uncommon frame below the target and treat that as the effective target.

3) Treat the frames that are at the target or below and are common to both the near and far outlines as forced explicitly re-exited and re-entered frames. This excludes frames that are above the target frame in the hierarchy.

After some use case analysis we decided that the last one, 3) was the preferred approach. What this means is that the transition to a frame that is already part of the active outline is specifying a *forced* transition to all frames below the target outline that are also in the current outline. All the frames from the target on down that are in the current active outline and hence also in the target outline will be included in the explicit re-exit and explicit re-enter list. The notion is that if a transition specifies a target that is in its active outline it was for the purpose of forcing the exit and enter actions to be re-executed. This is useful when some multi-iteration activity such as an auxiliary needs to be performed multiple times in succession. The exit and enter actions in these forced re-exit and re-enter frames will be explicitly executed as if the frames where not common to the near and far outlines. Although in a sense the forced frames are exited and re-entered, these forced exit and entry frames are not part of the implicitly re-enter and re-exit list of frames so the actions in the rexit and renter contexts for the forced frames are not executed. To restate, implicit re-exit and re-enter trigger the *rexit* and *renter* action contexts. Whereas explicit re-exit and re-enter does not but instead re-trigger the *exit* and *enter* contexts.

For example , if the current active outline is represented by the list of frames, `[a, b, c, d, e, f, g]` and the target of the transition is frame `c` with outline `[a, b, c, d, h, i]`, then a transition from any of the frames in `[a, b, c, d, e, f, g]` to frame `c` would produce

implicit re-exit/re-enter frames = `[a, b]`, explicit exit frames = `[e, f, g]`, explicit enter frames = `[h, i]` and forced explicit exit/enter frames = `[c, d]`. This is because with `c` as the target, frames `[c, d]` are part of the active outline, both before and after the transition. Functionally, with this approach, the forced exit/enter frames are treated the same as the non forced explicit exit and enter frames. So the transition consists of implicit re-exit/re-enter frames = `[a, b]`, explicit exit frames = `[c, d, e, f, g]` and explicit enter frames = `[c, d, h, i]`

Once the explicit exit, explicit enter, and implicit re-enter/re-exit frame lists have been computed, the frame then sequences through the list of explicit enter frames in top down order and evaluates the associated beacts of each frame until one of the beacts fails or all the beacts succeed. If the beacts are not all successful the transition fails and the framer starts executing the next precur. If all the beacts of a transition succeed, then the transition is successful and no more precur actions will be executed on this iteration of the framer. The framer then sequences through the list of explicit exit frames in bottom up order and executes the associated exacts. The framer then sequences through the list of implicit re-exit frames in bottom up order and executes the associated rexit actions. The frame then sequences through the list of implicit re-enter frames in top down order and executes the associated renter actions. The framer then sequences through the list of explicit enter frames in top down order and executes the associated enacts. The outline specified by the target frame is then made the new active outline. The transition is now complete.

Once all the precur processing has completed, either because no transitions were successful or after a successful transition is completed. The framer finalizes the iteration by sequencing through all the frames in the active outline in top down order and executes the associated *reacts*. This completes the iteration of the framer.

In simplified form the Framer execution order is as follows:

```
(Do one time)
CheckStart Framer
   Activate starting frame outline
   FOR EACH Frame in the active outline from the top down
      FOR EACH Beact in the Frame first to last
         IF Beact fails THEN
            Stop Framer and Return.

   OTHERWISE CheckStart Framer succeeded continue

Enter Framer (one time)
   FOR EACH Frame in the active outline from the top down
      FOR each Enact in the Frame first to last
         execute Enact

Recur Framer (one time)
   FOR EACH Frame in the active outline from the top down
      FOR EACH React in the Frame first to last
         execute React
Time Step

(Repeat until framer stopped)

Precur Framer
   FOR EACH Frame in the active outline from the top down
      FOR EACH Preact in the Frame first to last
         IF Preact succeeds (a transition has occurred) THEN
            stop processing Preacts
```

```
          ELSE
             continue with next Preact
Recur Framer
   FOR EACH Frame in the active outline from the top down
      FOR EACH React in the Frame first to last
         execute React
Time Step
...
```

When the preact is a Transition the execution of the preact and induced exacts and rexacts is as follows:

```
IF transition condition is False THEN
   return from Transition with failure

OTHERWISE condition succeeded continue

   compute lists of explicit exit, enter and implicit re-exit, re-enter frames
   FOR EACH explicit enter FRAME from top down
      FOR EACH Beact in Frame from first to last
         IF Beact fails THEN
            entry check failed return from transition with failure

   OTHERWISE entry check succeeded proceed with exit entry

   FOR EACH explicit exit Frame from bottom up
      FOR EACH Exact in Frame first to last
         execute Exact

   FOR EACH implicit re-exit Frame from bottom up
      FOR EACH Rexact in Frame first to last
         execute Rexact

   FOR EACH implicit re-enter Frame from bottom up
      FOR EACH Renact in Frame first to last
         execute Renact

   FOR EACH explicit enter Frame from top down
      FOR EACH Enact in Frame first to last
         execute Enact

Activate new active outline from target frame
Return from Transition with success
```

3.3.3. Auxiliary Framers

Each Frame may have zero or more Auxiliary Framers (Auxes). These Auxes are Framers in their own right and execute an associated framework of frames with actions. These frames may also have their own Auxiliary Framers and so on. This forms a nested hierarchy of Framers. The actions in each Aux are executed within the associated action context of the Auxes' Main Frame. In other words, for each Frame context such as enter, exit, recur, precur, etc, there is an associated Aux Framer activity that runs the actions of the same context for its associated frames and so on down the hierarchy. The arrangement is designed to allow subsequences or subsets of action frameworks to be reused conveniently within a higher level framework. Thus a very simple execution environment allows for sophisticated hierarchical composition of action frameworks.

3.3.4. Framer Execution Order with Auxiliaries

The Framer execution order including the execution of Auxiliary Framers within each Frame's activities is shown below:

```
(Do one time)
CheckStart Framer
   Activate starting frame outline
   FOR EACH Frame in the active outline from the top down
      FOR EACH Beact in the Frame first to last
         IF Beact fails THEN
            Stop framer and Return
      FOR EACH Aux in the Frame first to last
         IF CheckStart Aux fails THEN
            Stop Framer and Return

   OTHERWISE CheckStart Framer succeeded continue

Enter Framer
   FOR EACH Frame in the active outline from the top down
      FOR each Enact in the Frame first to last
         execute Enact
      FOR EACH Aux in the Frame first to last
         Enter Aux

   FOR EACH Frame in the active outline from the top down
      FOR EACH React in the Frame first to last
         execute React
      FOR EACH Aux in the Frame first to last
         Recur Aux

Time Step

(repeat until Framer stopped)
Precur Auxes
   FOR EACH Frame in the active outline from the top down
      FOR EACH Aux in the Frame first to last
         Precur Aux

Precur Framer
   FOR EACH Frame in the active outline from the top down
      FOR EACH Preact in the Frame first to last
         IF Preact succeeds THEN
            stop processing Preacts
         ELSE
            continue with next Preact
Recur Framer
   FOR EACH Frame in the active outline from the top down
```

```
       FOR EACH React in the Frame first to last
          execute React
       FOR EACH Aux in the Frame first to last
          Recur Aux

Time Step
...
```

When there are auxiliaries and the preact is a Transition the execution of the preact and induced exacts and rexacts is as follows:

```
IF transition condition is False THEN
   return from Transition with failure

OTHERWISE condition succeeded continue

   compute lists of explicit exit, enter and implicit re-exit, re-enter frames
   FOR EACH explicit enter FRAME from top down
      FOR EACH Beact in Frame from first to last
         IF Beact fails THEN
            entry check failed return from transition with failure

   OTHERWISE entry check succeeded proceed with exit entry

   FOR EACH explicit exit Frame from bottom up
      FOR each Aux in the Frame first to last
         Exit Aux
      FOR EACH Exact in Frame first to last
         execute Exact

   FOR EACH implicit re-exit Frame from bottom up
      FOR EACH Rexact in Frame first to last
         execute Rexact

   FOR EACH implicit re-enter Frame from bottom up
      FOR EACH Renact in Frame first to last
         execute Renact

   FOR EACH explicit enter Frame from top down
      FOR EACH Enact in Frame first to last
         execute Enact
      FOR EACH Aux in the Frame first to last
         Enter Aux

Activate new active outline from target frame
Return from Transition with success
```

Note that renter and rexit contexts for a frame do not affect the frame's auxiliaries since there is no meaningful concept of a rexit or renter auxiliary in and of itself.

3.3.5. Auxiliary Precur Actions

A note about the the execution order for auxiliary precur actions. There were two reasonable choices for the execution order for these. The first is to execute the Precur actions for each auxiliary just before or after the Precur actions of the aux's main frame. This seemed a natural way to do it since the other Aux actions happen the same way. The second , which we implemented, is to execute all the Precur actions of all the auxes in all the active frames before executing any precur actions in any of active frames.

The reason for selecting the second approach, although more complicated, is that it is more consistent

with the ioflo philosophy that higher level transitions should have priority over lower level transitions. This becomes a distinction when the transition is conditioned on the state of an auxiliary. As a result, one expects that a higher level transition should reflect the state of lower level auxes on the same framer iteration cycle. For example, if a higher level transition has as its condition the completion state of the a lower level auxiliary, then that should take priority over any lower level transitions that also have the aux completion state as a condition. Suppose when looking at the state of an aux versus time, that at time *t* the aux completed, one would expect to also see the highest level transition, that is conditioned on that aux completion, take priority and be successfully performed at time *t* as well. This is true for the second approach, but not the first. In the first approach, a lower level transition could be successful after the auxiliary completed on the same time step thereby preempting a higher level transition which would not see the auxiliary completion until the next time step. Even if there were no lower level transitions conditioned on the auxiliary completion, it is still disconcerting to expect the higher level transition to occur at time *t*, but not see it occur until time (*t + 1)*. One could argue that its not important to preserve this ordering, but in our testing we found ourselves feeling misled anytime the highest priority transition at time *t* did not take precedence and wondering what bad thing had happened.

3.3.6. Conditional Auxiliary Framers

A special precur action is a conditional auxiliary action. The purpose of a conditional auxiliary is to provide a way to execute repair activities as part of a reliable services envelope that allow the ioflo to suspend the execution of lower level frames until the repair is completed and then resume the active outline. A conditional auxiliary performs a similar role to an exception handler or interrupt service routine but does it in a way that is compatible with the ioflo paradigm.

As previously mentioned, a conditional auxiliary includes a condition and an auxiliary framer for the target. The action performs much of what the ioflo tasker does for running a Framer or what a Frame does when running a non-conditional auxiliary. When the associated condition is False then the action fails and returns false. If the condition is True then the action tries to start running the auxiliary. First, the beginning outline for the auxiliary is activated. Then the auxiliary checks to see if all the beacts in the frames of it's active outline are successful. This is done by sequencing through the frames in it's active outline in top down order. The beacts in each frame are executed until either one returns false or all succeed. An additional check is made to ensure that no other Frame is running this auxiliary. If so then the the auxiliary can't be started and the preact fails.

If one of the beacts is false then the auxiliary can't be started and the preact fails. If all are true the auxiliary is started. The auxiliary performs the first iteration for a Framer as described in the previous sections. The active outline for the Framer running the main Frame of the conditional auxiliary action is truncated to remove all frames below the Main frame. This effectively suspends the operation of the lower level frames. The preact sets a flag to indicate that it has been activated and returns true. On each subsequent execution of the preact, the auxiliary framer is iterated as per the repeated iteration order for Framers described in the previous sections above, until it is done. As long as it is not done, the preact returns true. Once the auxiliary is done, the preact restores the full active outline to the Framer running its Frame and returns false. This resumes the execution of lower level frames in the preact's Frame's Framer.

3.4. Code Objects

What follows is a brief overview of some of the Python code objects. A simple diagram of the ioflo software object architecture is shown below.

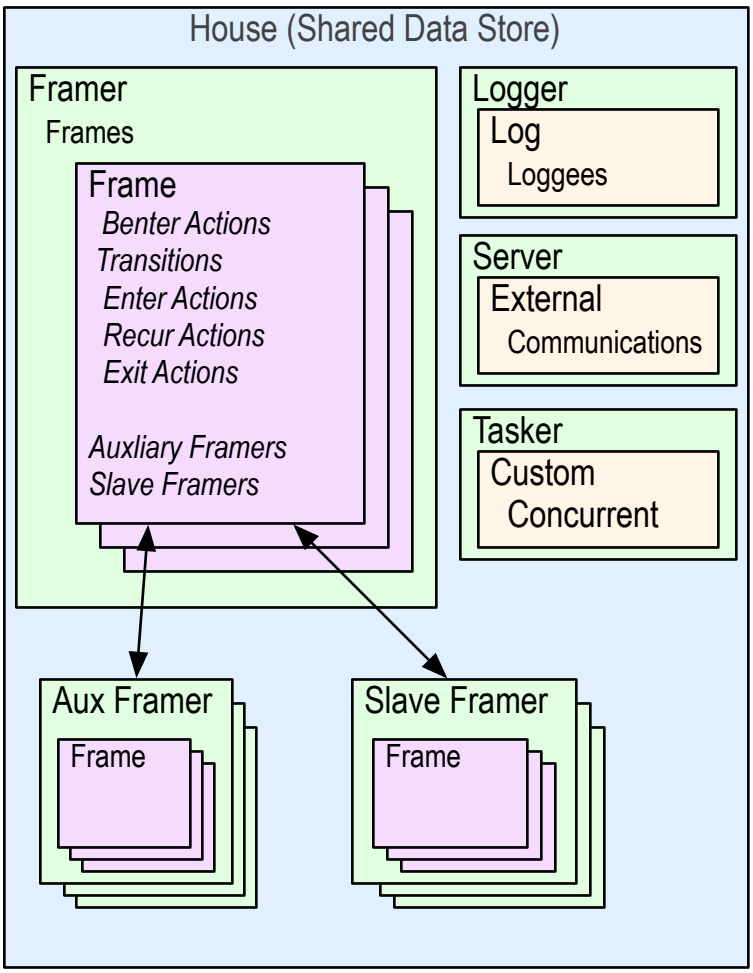

Fig.3.2: ioflo Components

3.4.1. Builder

The Builder object is responsible for loading and parsing the floscript files for a mission then creating the associated Houses. Without floscript, a HAF could be built directly using Python code. The purpose of floscript was to automate the build process in a user friendly way that significantly reduces apparent complexity. But there is nothing to prevent someone from using the HAF without floscript. In this case the Builder would not be used.

3.4.2. Skedder

A Skedder is the object that runs the tasks in a list of Houses. A Skedder runs Tasker objects via their generator. The generator is an example of a weightless thread. The Skedder has a period attribute that determines how often the Skedder runs. The Skedder is the master scheduler for one or more Houses.

3.4.3. Registry

The Shares, Store, Tasks, Framers, Frames, and Actors are all subclasses of the Registry class. The Registry class includes a class wide database of instance names. This allows the Registry to ensure that every instance has a user friendly unique name for an identifier. This makes it easy to implement run time scripts for configuring Frameworks since all the references for actions, frames, etc can be uniquely resolved from the user friendly name. Indeed floscript relies heavily on the respective Registries for each object type to build a HAF.

3.4.4. House

A House object contains the shared data Store, one or more Frameworks/Framers, and one or more Tasks for a single autonomous vehicle or agent community.

3.4.5. Store

Each House has a shared data Store. This Store acts as a publish subscribe database for the House. All the Tasks and Framers in the House have access to the same Store. This enables all the objects to communicate and interoperate. The Store uses a hierarchical associative database. This makes it much easier to keep track of and manage the Shares in the Store. Each Store has a global time stamp that is used to synchronize the time of changes to Shares in the Store.

3.4.6. Share

Each entry into Store is a Share. Shares may be single or multi-valued. Each Share keeps a time stamp of when the data in the Share was last updated.

3.4.7. Tasker

A Tasker is an object that can be iteratively scheduled for execution via its generator by a Skedder. The generator is created by the Tasker's makeRunner method. The generator is executed with control values , READY, START, RUN, STOP, and ABORT and may be in one of five states, READIED, RUNNING, STARTED, STOPPED, and ABORTED. Each tasker has a period attribute that determines how often the Tasker should be run. There are several subclasses of Tasker. These include Servers, Monitors, Loggers and Framers.

3.4.8. Framer

A Framer Tasker executes a hierarchical action framework (HAF).

3.4.9. Logger

A Logger Tasker periodically retrieves data from Shares in the Store and saves the data to a file

3.4.10. Server

A Server Tasker periodically communicates over the network or serial port to exchange data with other entities. The Server reads and writes from the Store.

3.4.11. Frame

A Frame object has attributes and methods for referencing and executing and its Actions including its Auxiliary Framers. The core functionality of the HAF is provided by the Framer and Frame object implementations.

3.4.12. Actors and Acts

There is no Action object. Action is the generic term used to represent the combination of objects, methods and attributes associated with an Action in the HAF. This permits the use of the term Action to be used independently of a specific python object. The reason for this is that there may be multiple ways to implement an Action and the implementation may change to better optimize computing resources.

In the current HAF implementation there are several different objects that implement Actions. This was done to better manage memory and computation resources.

Consider for example, an Action expressed as, *Set the depth controller set-point to 5.0 meters*. Another similar Action would be, *Set the depth controller set-point to 10.0 meters*. Both Actions perform the same activity but with a different value for the set-point. In general there are a limited number of activities but a potentially infinite number of unique parameter values. The activities are relatively heavy weight because they must implement functional code. The parameters may be lighter weight, such as a data structure. All that is needed is another light weight object to associate the two. Consequently it makes sense, from a code size perspective, to define a single object instance (singleton) for each type of activity and then define a lighter weight object to store a reference to the singleton activity object and the specific parameters values.

The activity is coded as an Actor object. The parameters are coded as a Python dictionary. Each Actor object has a method named *action* that executes its activity on the passed in parameters. The lightweight object holding references to the Actor and parameters is an Act.

Each Act object has a reference to the associated activity Actor and also a reference to the specific parameter dictionary for the Action. An Act object can be referenced as a function call that executes it's Actor's action method on the parameters.

4. FloScript Overview

4.1. Python

A constrained configuration language that prevents coding mistakes in the stress prone operational environment is a vital capability. To achieve this capability we developed an domain specific declarative script language we call FloScript. FloScript is implemented in Python. Although, one could implement an IoFlo HAF with straight Python code, it would still be highly inconvenient to use and would not benefit from the reliablity resulting from a well constrained configuration script language. Moreover, because convenience of expression for potential non-expert users was a primary design goal, learning Python was seen as too much of a barrier. Furthermore we wanted to enable in field or production time reconfiguration. In real world operations, the success rate and cost effectiveness is significantly improved if behavior changes can be performed without re-compiling or even re-byte compiling.

4.2. Features

In order to support any reasonable execution and scheduling architecture we wanted support within FloScript for both nested and concurrent frameworks. We wanted support for multiple entities (e.g. platforms) in the same script so that we could simulate concurrent multiple platform operations easily. We wanted support for the multiple layers of hierarchy such as a hierarchy of frameworks and nested behaviors that are also state machines. We wanted a unified scheduling and execution environment, that, in this case is a cooperative weightless thread scheduler instead of some arcane combination of OS processes, OS threads and explicit schedulers. We wanted integrated logging and network communications. We wanted the ability to use combinations of reactive behaviors, and deliberative behaviors in the same framework. Finally we wanted support for multiple data stores. FloScript provides a way to manage and configure all of these features with a parsimonious set of declarations.

To our knowledge every autonomy or automation architecture eventually develops some form of simplified configuration method to avoid hard coding each and every mission. Often these configuration methods are developed in an ad hoc manner and consist of some combination of configuration files, shell scripts, make scripts, and mission script files, that oftern appears to have been put together as an afterthought. We decided that a unified approach where ALL the configuration is done in one place with one script language is the best way to reduce apparent complexity. The goal was to produce one unified script to configure and run everything. We believe FloScript goes a long way to accomplishing this goal.

This required a significant amount of up development effort on both the configuration script and the underlying architecture. Ioflo's HAF and FloScript are being developed in parallel. FloScript provides a convenient way to build Houses and their associated Stores, Taskers, Framers, Frames, Actors and Acts. FloScript also allows the construction of multiple Houses in a single script and also the loading of multiple files in the composition of a script. To use a "Lord of the Rings" analogy we wanted:

One script to write them all, One script to build them,
One script to bind them all and conveniently run them.

4.3. Contexts Revisited

FloScript is a line oriented contextual declarative configuration language. This is different from an imperative block oriented language or even a declaritive block oriented language. A contextual language uses context change as the primary organizing method. Using context change as the primary organizing method for grouping declaration expressions minimizes syntactic elements.

There are two types of contexts used in FloScript. The first are *action execution contexts*. The second are *lexical grouping contexts*. The action execution contexts or '*actioning*' or '*acting*' contexts were described in some detail in a previous section.

Another way of thinking about the *lexical grouping contexts* is that they define a *frame of reference* for the action execution contexts. This *frame of reference* concept provides one motivation for the FloScript declaration naming scheme for the principle contextual grouping commands, `frame` and `framer`. The *frame of reference* is specifically related to the ordering of declarations in FloScript. Because *frame of reference* and *lexical grouping* are such a cumbersome and potentially ambiguous terms, the shorthand *framing* or *framed* context will be used instead. The primary purpose of some FloScript declarations is to define a *framing* context that will get executed and scheduled. The

primary purpose of other declarations to to define individual actions that will be executed during an *actioning* context within the enclosing *framing* context.

FloScript declarations are not themselves executed but create actions that are framed and executed by context. Lexical order is important since the actions resulting from declarations that are in the same framing context will be framed and/or executed in the order they appear in the FloScript. This gives a consistent reproducible framing and priority ordering to all the actions within a given context.

Framing context and change of framing context is implied by the declaration. This means that there are no merely syntactical context delimeters in the language. For example a `frame` declaration starts a new Frame context. The next `frame` declaration ends the previous Frame context and starts a new Frame context.

Framing contexts may be nested. This means that at a given contextual level, a framing context change ends not only the current context at the current level but also ends all nested contexts at all lower levels. For example, a `framer` declaration starts a new Framer context. Any subsequent `frame` (not `framer`) declarations will be nested within the current Framer context. Likewise any `do, set, put, go` action declarations that follow a `frame` declaration will be nested within the current Frame context. A new `framer` declaration will end, not only the current Framer context but will also end the current Frame context as well. This is illustrated in the diagram below.

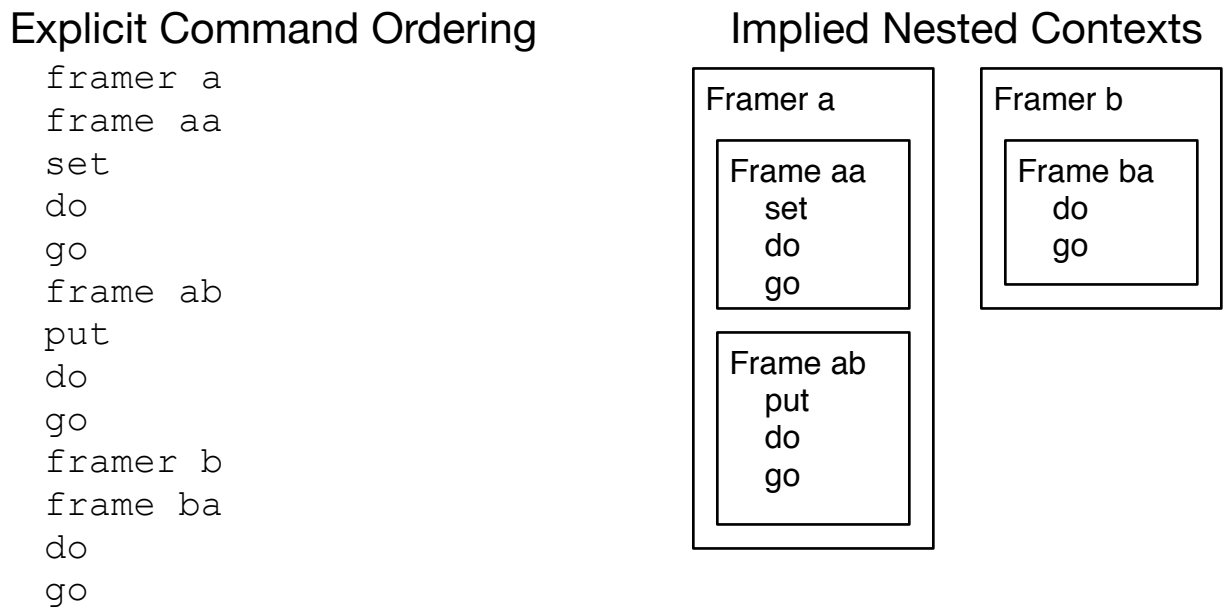

Fig.4.1: Context Nesting

As discussed previously Frames themselves may be nested within other Frames and Framers may be nested within other Framers. How FloScript accomplishes this will be discussed later.

4.4. Declaration Syntax

Each declaration line is delimited by a newline character '\n' and defines a single FloScript declaration. Longer declarations may be continued across multiple lines by escaping the newline character with with a backslash ('\\\n').

declarations consist of words (tokens) separated by whitespace. The simplest analogy is to a sentence with a verb, object and prepositional phases. This serves to minimize syntactic elements. The first word is the verb that idenfies the type of declaration. Any other words that follow the declaration verb may be separated by one or more whitespace characters. Words may be arguments or prepositional connectives. Although the words in the declaration are delimited by white space, extra white space between words or at the beginning or end of the line is not significant. Thus declarations may be indented with tabs or spaces for more readability but the indentation does not change the interpretation of the declaration.

Connectives are reserved words typically prepositions that indicate the purpose of the following target parameter. Connectives tend to be prepositions that evoke the purpose of the target parameter. For example, the connectives used by various commands include the following:

```
'to', 'with', 'by', 'from', 'per', 'for', 'as', 'at', 'in', 'of',
'on', 'if', 'be', 'into', 'and', 'not', '+-', '==', '<', '<=',
'>=', '>', '!='
```

Parameters delimited by double quote characters may include white space between the quotes. For example *"Hello world"*.

To restate, a declaration starts with a declaration verb followed by an object and prepositional phrases containing parameters. FloScript is case sensitive, so `BigDog` is not the same as `bigdog` for a verb, connective or parameter. FloScript only uses lowercase for verb tokens and reserved tokens such as connectives. The argument variables may be different cases as desired.

Comments are indicated by the `'#'` character. All characters on a line that follow a # char are part of the comment and are ignored. Comments may start a line or appear within a line. Any characters after a comment character are ignored.

The FloScript declaration definitions that follow use `courier monospaced font`. When an element is *italicized* it means that it is a variable whose value will be substituted later. When it is not italicized then the exact text is required. An ellipsis … means the preceding element may be repeated. Elements contained in square brackets [] are optional. Parentheses () surround a selection list of comma separated choices. One and only one of the elements in the list must be used. A term definition is given by the name of the element and a colon, as in, `Term: definition`. The different forms of the element are provided below the term, each on a separate line.

Following the conventions above, declaration tokens and reserved argument tokens are `regular font style`. Argument variables are in *`italic`*. A choice between two or more required arguments is indicated by listing the choices in parenthesis, such as, `(change, update)`. Optional arguments are surrounded by square brackets, such as, `[of frame [framename]]`.

For example, the following two declarations are equivalent. The second has a line continuation and a comment.

```
go next if depth >= 10.0 and speed >= 5.0

go next if depth >= 10.0 and \
```

```
    speed >= 5.0 # a comment here
```

For example the following declaration is commented out.

```
#This line is commented out
```

The following declaration ends with a comment.

`go next if depth >= 10.0 and speed >= 5.0` # this is a comment

The following elements are common to many of the definitions and are provided here to avoid redundancy.

dotpath:
   [*.node*[*.node* ...]].*share*

path:
   [*node*.[*node*. ...]]*share*

node:
   identifier

share:
   identifier

fields:
   field [*field* ...]

field:
   identifier

identifier:
   letter[(*letter*, *digit*, *underscore*) ...]

value:
   (string, *boolean*, *number*)

string:
   "characters" any character except " between quotes

boolnum:
   (*boolean*, *number*)

boolean:
   (true, false) not case sensitive

number:
   (int, hex, octal, float, complex)

5. Address Modes

5.1. Overview

The FloScript declarations use several different addressing modes to reference and access data that is published in the Store. Since the Pub-Sub Store is the central element of almost all activity in *ioflo*, a flexible well defined mechanism for accessing and transferring data in and out of the Store is vital. Each element in the Store is called a Share and may have one or more values. The underlying Python implementation of a Share is an object that is similar to an ordered dictionary (ordered associative array) but with enhanced functionality to support ioflo. Each Share holds a Data record. The record consists of one or more field, value pairs. When a record is created with multiple values, each must have a unique field string (key). Thus each data value in each Share has a corresponding key or field. The default field when none is provided is the field denoted 'value'.

The Store uses a hierarchical associative database to store the Shares. A hierarchical associative database can be visualized as a tree root like system. The base or root is at the top with more root branches extending downwards. Branches split at nodes into one or more additional branches. In *ioflo*, the tip of each branch is where a Share is stored. The branch splits are Nodes which are ordered dicts (associative arrays). The keys of each Node dict are strings of characters, the values are Shares or other Nodes. The Root is the top level Node. Every Node but the Root is a value referenced by a key in the Node above. The Share at the tip is also referenced by a key in the Node above it. Each Node or Share may be uniquely addressed with an ordered list of keys. Starting at the top, the hierarchy is descended recursively by retrieving the value associated with the next key in the list.

FloScript represents this list with a string in dotted notation where each key is separated by a dot. The Root has no key so it is represented with a leading dot. This dotted notation string is called a *Path*. For example, the string, "*.scenario.origin*" is the full path for a share referenced with key = "*origin*" in a Node referenced with key = "*scenario*" in the Root Node. Each Share stores its *Path* string as its name. Thus a Share may be looked up in the Store via its name. Because "path" can have other meanings, when necessary for the sake of clarity, we sometimes use the term *path name* or *pathname* instead.

The purpose of the a hierarchical tree of nodes is to make it easy to organize entries in the Store by grouping related Shares together in namespaces, much like a file system with nested directories. For example, the share with path name "*.scenario.origin*" could be described as the share *origin* attached to the *scenario* node attached to the *root* node. Likewise the path name "*arbiter.depth.parm*" could be described as the share *parm* attached to the *depth* node which is attached to the *arbiter* node attached to the *root* node. Often, it is more convenient to refer to a Share or Node, not by its full path name, but only by the last key in its path name, much the way a file name is used to reference a file in a given directory.

Because there is only one root, converting the path name to a list provides the same information with or without the leading dot. FloScript, however , uses the presence or absence of the leading dot to remove ambiguity when parsing relative address modes where nodes in the path might be implied and thus the full path may not be provided explicitly in the FloScript declaration.

In general the path name is not used to reference a share except at FloScript parse time. Once the script is parsed, the Shares are referenced directly in the Actions by their memory address. However an external network connection to the Store might need to use the path name. Associative array

access using hash tables as per the Python implementation are actually quite fast relative to network latencies for transmitting data.

5.2. Addressing Modes

The two main modes of addressing are direct and indirect. Direct means the data fields and values are provided directly in the declaration. Indirect means that the data fields and values are retrieved from a Share in the Data Store.

There are several different Indirect addressing modes. The two primary indirect addressing modes are absolute and relative.

Absolute means the Share is referenced with the complete path name of the Share in the Store. Relative means that the Share is referenced by a partial path relative to some other location in the Store. The relative location may be either explicitly provided in the declaration or implied by the type of the declaration.

The syntax for the different addressing modes is provided below.

```
address:
   direct
   indirect
```

5.3. Direct

```
direct:
   [value] value
   field value [field value ...]

directone
   [value] value
   field value
```

The difference between `direct` and `directone` is that `direct` allows for multiple valued data whereas `directone` only allows for single valued data. The `directone` address mode is necessary, because some declarations require at most a single value.

5.4. Indirect

```
indirect:
   absolute
   relative
```

Each of the indirect addressing modes may be preceded by an optional field selector list. The field selector consists of a list of field names separated by spaces and followed by the connective `'in'`. The field selector appears before the indirect address of the share. By default, when a declaration references a share via an indirect address and the field selector list is not provided, it implies that all the pre-existing field or fields in the share are to be used. If a field selector list is provided then only those fields in the field selector list are used. In some declarations only a single field is allowed so only a single element is in the field selector list. When only a single element is allowed the term is

singular such as `value` or `field`. When multiple elements are allowed, then the term is plural, `fields`. For example:

```
[(value, field, fields) in ]
fields:
   name1 name2 ...
```

When a declaration is parsed that references a share, and there are no pre-existing fields in that share, then a default field or fields are created for that share based on the rules below.

The field selector list is denoted by a *`fields`* element which may consist of one or more field names. When *`fields`* is missing then *`fields`* is resolved by substitution according to the following general rules (unless overridden by declaration specific rules):

When the addressed share is the source of the data to be transferred these rules apply.

IF the addressed share has preexisting fields THEN

IF the addressed share has a field = '`value`' THEN

use '`value`'

ELSE IF the data is being copied into another share where fields are provided THEN

use the provided fields

ELSE

raise a parse error that the source fields are indeterminate

ELSE

use '`value`'

When the addressed share is the destination of the data to be transferred these rules apply.

IF the addressed share has preexisting fields THEN

IF the addressed share has a field = '`value`' THEN

use '`value`'

ELSE IF the data is being copied from direct data THEN

use the fields in the direct data

ELSE IF the data is being copied from another share where fields are provided THEN

use the provided fields

ELSE

use '`value`'

Although is is permitted that a single valued data Share use some field other then `value` it is not recommended. In addition, a multi-valued Share should not have a *`field`* = `value`. If a

declaration tries to assign multi-valued data to a pre-existing Share with the field = 'value' or vice versa, it will cause a parse error. Multiple fields in one Share all have the same time stamp and are logged together. Consequently, it is suggested that a Share have multiple fields only when the fields are updated at the same time or are rarely changed.

In some of the declarations, both source and destination indirect addresses may be provided for transferring the data. In the event that a field list is provided for both the source and destination then the number of fields must be the same in both lists and the order that the fields appear in each list is meaningful because the data is copied in the order from the first source field to the first destination field and not to the field with the same name unless it is in the same position in both lists. This enables copying between different fields in different shares.

The rules for resolving indirect relative addresses are as follows:

If the path begins with a dot "." and there is no following relation clause, then it is absolute addressing.

If the path begins with a dot "." and there is a following relation clause, then it is relative addressing without implied variants. In other words the combination of `dotpath` and a relation clause overrides any implied variants.

If the path does not begin with a dot then it is relative addressing and any implied variants are in force. The type of relative addressing is determined by the optional relation clause. If there is no following relation clause then by default root relative addressing is applied. Otherwise, root, framer, or frame relative addressing is applied as indicated by the relation clause.

5.4.1. Indirect Absolute

The primary component of indirect absolute addressing is the share path in dotted notation. The path must start with a dot. The dot indicates that the path starts at the root of the Store hierarchy and is therefore not relative to some branch lower down in the hierarchy. The initial dot also removes any ambiguity when parsing. The other component is an optional field list. The field list may have one or more field identifiers. If more than one field is provide then none of them may be 'value', If no field list is provided then the default field list is the single field = 'value'. A field list with the single field = 'value' is also valid. The connective 'from' may be used to directly indicate that the following token is a path. When a field list is provided then the field list comes first, followed by the connective '`from`' and then the path.

```
absolute:
   dotpath
```

Show below are examples of indirect absolute addresses using the connective `from dotpath`.

```
from .goal.depth
from .controller.pid.parms
from .filter.salinity.out
from .scenario
```

Depending on the declaration, a field selector with absolute addressing may be allowed.

5.4.2. Indirect Relative

There are three basic relative addressing modes, root relative, framer relative, and frame relative. For each basic relative addressing mode there may be one of two implied variants, "state" and "goal". Root relative addressing is for global information that is not Framer or Frame specific. The implied variants are defined below. (See page 34)

One motivation for the relative addressing modes is convenience of expression. The Store has predefined locations that are reserved for the use of various declarations and actions. Although, absolute addressing could be used, because these predefined locations are frequently referenced, it is less cumbersome to use a relative address than to always write out the full path for an absolute address.

Another motivation is to allow dynamic addressing. Some of the predefined locations are specific to the lexical context of the current Framer or Frame. Relative addressing allows one to address a location relative to the current Frame or Framer without having to provide the framer's or frame's name.

The third motivation for relative addressing is to allow the expression of implied variants. The implied variants are declaration verb specific, that is, a given verb will imply whether or not a particular implied variant is in force. There is no syntax to specify an implied variant, it is, as its name suggests, merely implied by definition for a given verb. For example, the `set` verb uses the implied "goal" variant. The `if` connective Need may use the implied "state" variant on the left hand side of a comparison and the implied "goal" variant on the right hand side. The majority of transition preconditions consist of comparing a "state" to a "goal" to see if the goal has been met before transitioning to another Frame. The implied variants make it much more convenient to compactly address the states and goals.

The actual addressing mode used is determined by three factors: the declaration expressing the address, the presense or lack thereof of a leading dot in the path, the presense of a relation clause. The different combinations of these three factors define all the addressing modes.

The generic form of relative addressing is a path followed by a relation clause. The relation clause is indicated by the "`of`" connective. Multiple relation clauses may occur for some times of relative addressing.

```
relative:
   path [of relation][of relation]
   dotpath [of relation][of relation]

relation:
   root
   framer [name]
   frame [name]
```

We label these three forms as follows:

```
relative:
```

```
root
framer
frame
```

The difference between the dotpath and path formats is that dotpath must have a leading dot and path must not. The reason for making a semantic distinction between the two is to allow an override of an implied variant on a declaration that uses an implied variant. A dotpath overrides the implied variant. To clarify, implied addressing variants can only occur with non `dotpath` addresses

Examples of the `dotpath` format follow:.

```
.goal.depth
.controller.pid.parms
.filter.salinity.out
.scenario
```

Examples of the `path` format follow:

```
goal.depth
controller.pid.parms
filter.salinity.out
scenario
```

5.4.2.1. Root Relative

Root relative addressing means that the path provided is relative to the root or first level of the Store hierarchy. By default if the relation clause is missing on a non *`dotpath`* path then root relative is assumed. At first glance root relative addressing seems to be the same as absolute addressing since the path to the share is anchored to the root of the Data Store hierarchy. This is true except when a declaration that uses one of the implied variants is specified. The presence of a *`dotpath`* prevents the implied variant if any from occurring. So a *`dotpath`* with a root relation clause is the same as absolute addressing. Therefore without loss of generality, if there is no implied variant, root relative addressing is equivalent to absolute addressing.

```
root:
   path [of root]
   dotpath of root
```

Without implied varients, in both cases, the share path resolves to *:*

```
.path
```

5.4.2.2. Framer Relative

There are certain data values in the Store that are associated with a given framer. One might wish to store a counter of some other bookkeeping information about a specific framer. The convention for storing these is to place them off from the branch `.framer.`*`name`*`.` where name is the name of the framer. For example, the one might wish to store a counter that is specific to framer "mission". This

could be placed in `.framer.mission.counter`. As with root relative addressing there are two implied variants, state, and goal. But in this case, state is a framer specific state and goal is likewise a framer specific control reference. For example, `.framer.name.state.elapsed` holds the elapsed time that the framer has been executing the current outline and `.framer.name.goal.elapsed` holds the desired duration for evaluating the current outline. Using an absolute path (dotpath) with an `of framer` clause will do framer relative and override any implied variants.

```
framer:
   path of framer [name]
   dotpath of framer [name]
```

Without implied varients, in both cases, the share path resolves to*:*

`.framer.name.path`

When `name` is missing then the name of current framer in the lexical context of the script is used.

5.4.2.3. Frame Relative

Frame relative addressing is similar to framer relative except that its relative to a given frame which is also relative to the associated framer for that frame. There are certain data values in the Store that are associated with a given frame. One might wish to store a counter of some other bookkeeping information about a specific frame. The convention for storing these is to place them off from the branch `.framer.framername.frame.name.` where framername is the name of the framer that holds the frame, and name is the name of the frame. For example, the one might wish to store a counter that is specific to frame "surface" in the framer "mission". This could be placed in `.framer.mission.frame.surface.counter`. As with root and framer relative addressing there are two implied variants, state, and goal. But in this case state is a frame specific state and goal is likewise a frame specific control reference. Using an absolute path (dotpath) with an `of frame` clause will do frame relative and override any implied variants. Because frame relative is also framer relative, two relation clauses can appear.

```
frame:
   path of frame [name][of framer [framername]]
   dotpath of frame [name][of framer [framername]]
```

The share path resolves to*:*

`.framer.framername.frame.name.path`

When `name` is missing then the name of current frame in the lexical context of the script is used. If the `[of framer ...]` relation clause is missing then the current framer in the lexical context is used. If the `[of framer ...]` relation clause is provided but `framername` is missing then the `framername` is given by the lexical context of the current framer.

5.4.3. Indirect Implied Relative

As previously mentioned, currently, ioflo supports two implied variants of all three relative addressing types. These are "state" implied and "goal" implied relative addressing.

The node named `state`, or in absolute addressing syntax `.state`, is defined as the location where data that represent the state of the platform is stored. This is a necessary convention that enables the actions to know where to look for state information. For example (using absolute addressing syntax) in `.state.heading`, `.state.depth`, and `.state.speed`, are stored respectfully the heading depth and speed of the platform. The state implied variant of relative addressing inserts `.state` into the path. This means one can just use `heading`, `depth`, or `speed`, instead of the more tedious `state.heading`, `state.depth`, or `state.speed` to address these shares. This also means that in the future the implied node location could be changed without changing the syntax of FloScript.

The node named `goal`, or in absolute addressing syntax `.goal`, is defined as the location where data that represent the declaration and control goals, such as desired set-points or controller references for the platform are stored. This is a necessary convention that enables the actions to know where to go for goal information. For example (using absolute addressing syntax) in `.goal.heading`, `.goal.depth`, and `.goal.speed`, are stored respectfully the desired heading depth and speed of the platform for the autopilot. The goal implied variant of relative addressing inserts `.goal` into the path. This means one can just use `heading`, `depth`, or `speed`, instead of the more tedious `goal.heading`, `goal.depth`, or `goal.speed` to address these shares. This also means that in the future the implied node location could be changed without changing the syntax of FloScript.

5.4.3.1. Implied Root Relative

There are three forms of root relative addressing, these are, root, rootstate, or rootgoal. The last two are implied variants.

The "state" implied variant of root relative addressing has `.state` is inserted in the path and

`path [of root]`

resolves to:

`.state.path`

The "goal" implied variant of root relative addressing has `.goal`. is inserted in the path and

`path [of root]`

resolves to:

`.goal.path`

Which one of the three forms of root relative addressing that is applied when `of root` appears in the address is dependent on the particular FloScript declaration. Some declarations use the implied variants, rootstate or rootgoal, and some do not. See the documentation of each declaration.

5.4.3.2. Implied Framer Relative

There are three forms of framer relative addressing, these are, framer, framerstate, or framergoal. The last two are implied variants.

The "state" implied variant of framer relative addressing has `.state` is inserted in the path

`path of framer [name]`

resolves to:

`.framer.`*`name`*`.state.`*`path`*

The "goal" implied variant of framer relative addressing has `.goal.` is inserted in the path and

`path` `of framer [`*`name`*`]`

resolves to:

`.framer.`*`name`*`.goal.`*`path`*

Which one of the three forms of framer relative addressing that is applied, when `of framer` appears in the address is dependent on the particular FloScript declaration. Some declarations use the implied variants, framerstate or framergoal, and some do not. See the documentation of each declaration.

5.4.3.3. Implied Frame Relative

There are three forms of frame relative addressing, these are, frame, framestate, or framegoal. The last two are implied variants. Since each frame is unique to a given framer, frame relative addressing is resolved relative to the associated framer which is given by the framing context of the frame. This provides the *`framername`*.

The "state" implied variant of frame relative addressing has `.state` inserted in the path

`path` `of frame [`*`name`*`] [of framer [framername]]`

resolves to:

`.framer.`*`framername`*`.frame.`*`name`*`.state.`*`path`*

The "goal" implied variant has `.goal.` inserted in the path and

`path` `of frame [`*`name`*`] [of framer [framername]]`

resolves to:

`.framer.`*`framername`*`.frame.`*`name`*`.goal.`*`path`*

Which one of the three forms of frame relative addressing that is applied, when `of frame` appears in the address is dependent on the particular FloScript declaration. Some declarations use the implied variants, framestate or framegoal, and some do not. See the documentation of each declaration.

6. Action Types

In order to better understand the FloScript declaration definitions, a couple of detailed preliminary concepts must be covered first. These are Addressing Modes and Action Types.

As previously stated, for the sake of clarity, in the FloScript exposition that follows, any verbatim element of a FloScript declaration is presented in `courier monospaced font`.

6.1. Action Contexts

As mentioned previously, each Action is executed in one of several action execution or *actioning* contexts. The contexts are *Benter, Enter, Recur, Exit, Precur, Rexit and, Renter*. Each Action type described below has a native or default context that is used for the Action unless otherwise overridden. Moreover, each FloScript action declaration implies an execution context. The native

context can be overridden for some Action Types so that these actions can be executed in different contexts. The context overriding declarations are `native, benter, enter, exit, precur, recur, rexit, and renter.`

6.2. Criteria - Needs

The short word is Need. A Need is a condition or criteria for doing something. Needs evaluate to true or false. Needs are used as conditions (or guards) for entering a Frame, or as conditions for a transition between Frames, or for conditional Auxiliaries. Needs are defined with the `if` floscript connector as part of other floscript declarations such as `go, let, aux` . For example:

```
go target if needs

let me if needs

aux name if needs.
```

More detail is provided in the individual declaration definitions for go, let, and aux below and in section 7.10 Need Syntax.

6.3. Explicit Data Store - Poke

The short word is Poke. A Poke action explicitly sets the value of a share's data. This is used in cases where some very specific value must be set explicitly as opposed to being set implicitly by some other action or behavior. Poke allows explicitly setting a value in a share by name. The three Poke declarations are `put`, `copy`, and `inc`. The Native context for all the Poke actions is enter. These declarations allow manipulation of data share items directly in FloScript. This really helps in developing mission concepts or one-off activities in a transparent manner. The native context of actions associated with `put`, `copy`, and `inc` declarations may be overridden.

6.4. Objectives -Goals

The short word is Goal. A Goal action sets a desired objective or set-point for a controller. For example, a goal action would fix a set point like, set heading to 50 degrees or set speed to 2 m/s. The goal declaration is `set`. The native context for the Goal actions is Entry, that is, upon entry to the frame set up all the goals to be sought after while in the frame. The context of actions associated with `set` declaration may be overridden.

6.5. Behaviors - Deeds

The short word is Deed. A Deed action runs a software algorithm or controller that processes inputs and internal state to produce outputs. For example the heading autopilot would be a Deed. There is one Deed declaration verb, `do`. The native context for actions associated with the `do` declaration is Recur. The context of actions associated with `do` declaration may be overridden.

6.6. Configurations - Traits

The short word is Trait. A Trait action sets up the configuration of one or more other aggregating Deed actions, such as Arbiters. For example a Track-Line trait would configure the autopilot to use track line control instead of homing control. The Trait declaration is `use`. The native context for actions associated with use is Entry but may be overridden. Trait actions are not yet implemented.

7. Basic Declaration Verbs

7.1. init

```
init destination (to, with) data

destination:
   absolute
   path

data:
   direct

init destination (by, from) source

destination:
   [(value, fields) in] dotpath
   [(value, fields) in] path

source:
   [(value, fields) in] dotpath
   [(value, fields) in] path
```

The `init` declaration can exist outside of a framer or frame lexical context, therefore framer and frame relative modes and the 'of' relation clause are not supported. This declaration has no implied variants for its path relative addressing. Source data is accessed at parse or build time not run time. Source must be pre-existing.

The `init` declaration creates a new entry in the Store of name *dotpath or path*.

In the `to` form, the initial field names and associated values given to the *destination* share are explicitly given by *field value* pairs, where *field* is the name of the field and *value* is the associated value assigned to that field. If there is only one token after the connective *to*, then the default field name ″value″ is assumed.

In the `from` form the values come from a pre-existing share given by the fields and path in *source*. This allows one to initialize a share by copying from another share. If the fields are not provided then the default rules apply.

In either form, the values in the fields in destination are overwritten if already existing and created and initialized otherwise.

Examples:

```
init .scenario.bottom to 15.0
init scenario.current with north .5 east -.5
init test.bottom from scenario.bottom
init north east in test.current by north east in scenario.current
init north east in test.current from scenario.current
```

```
init test.current from north east in scenario.current
```

7.2. put

```
put data into destination

data:
   direct

destination:
   [(value, fields) in] indirect
```

The native context is *enter* but can be overridden. This declaration has no implied variants for its relative addressing.

Put creates an Action that sets the values of the fields in the *destination* Share to the provided *data*. The associated Act references an instance named pokeDirect of the DirectPoke class.

Examples:

```
put 5.3 into test.big
put value 5.3 into test.big
put north 0.0 east 50.0 into scenario.origin
put north 50.0 east 40.0 into north east in scenario.origin
put 10 into counter of frame
put 10 into value in counter of frame
```

7.3. copy

```
copy source into destination

source:
   [(value, fields) in] indirect

destination:
   [(value, fields) in] indirect
```

The native context is *enter* but can be overridden. This declaration has no implied variants for its relative addressing.

This declaration creates an Action that sets the values of the fields in the *destination* share from fields in the *source* share. The associated Act references an instance named pokeIndirect of the IndirectPoke class.

Examples:

```
copy state.depth into test.depth
copy north east in origin into scenario.offset
copy elapsed in state of framer into test.elapsed
```

7.4. inc

```
inc destination (to, with) data
inc destination (by, from) source


destination:
   [(value, fields) in] indirect

data:
   direct

source:
   [(value, fields) in] indirect
```

The native context is *enter* but can be overridden. This declaration has no implied variants for its relative addressing.

The `(to, with)` form creates an Action that increments the values of the fields in the *destination* share by the values of the fields in the provided *data*. The associated Act references an instance named incDirect of the DirectInc class. To avoid potential ambiguity of the to option use the with option instead..

The *from* form creates an Action that increments the values of the fields in the *destination* share by the values of the fields in the *source* share. The associated Act references an instance named incIndirect of the IndirectInc class.

If multiple fields are provided then it performs a vector increment.

Examples:

```
inc counter of frame with 1
inc .test.counter with -1
inc test.turn from right in box
```

7.5. set

```
set goal (to, with) data
set goal (by, from) source

goal:
   elapsed
   recurred
   [(value, fields) in] indirect
```

```
data:
   direct

source:
   [(value, fields) in] indirect
```

The native context is *enter* but can be overridden.
This declaration uses the implied goal variant of relative addressing for the *goal* but not the *source*.
There are also two special implied variant cases, `elapsed` and `repeat`:

When goal is `elapsed` then the share at `framer.`*`name`*`.goal.elapsed` is used for the goal, where *`name`* is the name of the current framer.

When goal is `recurred` then the share at `framer.`*`name`*`.goal.recurred` is used for the goal, where *`name`* is the name of the current framer.

To override the implied variants including the special cases use `dotpath` form for the *goal* path.

The *to* form creates an Action that sets the values of the fields in the *destination* share to the values of the fields in the provided *data*. The associated Act references an instance named goalDirect of the DirectGoal class.

The *from* form creates an Action that sets the values of the fields in the *destination* share from the values of the fields in the *source* share. The associated Act references an instance named goalIndirect of the IndirectGoal class.

Examples:

```
set elapsed to 5
set recurred with 2
set depth to 15
set origin to lat 40.2345 lon 80.3456

set heading from .box.right
set heading by left in box
```

7.6. do

Do declarations have the following format:

```
do kind [part ...] [as name [part ...]] [at context]
   [to data] [by source]
   [per data] [for source]
   [with data] [from source]

deed:
   name [part ...]

kind:
```

```
   name [part ...]

context:
   (native, benter, enter, recur, exit, precur, renter, rexit)

data:
   direct

source:
   [(value, fields) in] indirect
```

Creates new Deed of type *kind* modified by adjective parts with optional name *name* modified by adjective parts. The modifiers create camel case identifiers. If *name* is not given then the name is the Deed type. This is actually the reverse camel case name of the underlying Python subclass.

The native context is *recur* but can be overridden. Creates a deed action where the deed action is specified by the *kind* plus optional *modifiers*. Internally the kind plus modifiers are used to lookup hte class and create a unique instance of the deed. This provides a hierarchical namespace for deeds. For example the declaration:

```
do controller pid speed
```

results in a *deed* action named *controllerPidSpeed*. This scheme allows name spacing of deed behaviors into logical groups to make it easier to manage. For a three part name think of the first part as the major type deed (controller), the second part as the minor type of deed (PID), the the third part as the specific type of deed (speed).

A deed is some type of behavior like a controller that executes repeatedly every time the framer runs (recurs). Much of the work in developing an autonomous control system is in creating the behaviors (deeds).

Some deeds also create implicitly a special restarter entry action that restarts the deed. This is useful for zeroing out integrators or other startup conditions.

Some deeds that have been created include: simulator, controller, filter, observer, arbiter.

For the controller type some more specific deeds that have been created are `pid, motion`. And for the pid type some specific kinds are `heading, depth, pitch, speed.`

The deed is usually completely specified by its kind. Example:

```
do controller pid heading
```

The option `at` allows specifying the actioning context of the deed without changing the current context. Example:

```
do deed at enter
```

The option `to` allows injection of direct data into the Deed's action as parms at link/run time to the deed's action method. The connective `to` indicates that direct data follows. Example:

```
do deed to depth 5
```

The option by allows injection of indirect data from a location in the store into the deed's action method as parms at link/run time. The connective by indicates that indirect data follows. Example:

```
do deed by depth in .bottom.depth
```

The option per allows injection of direct data into the initialization of the I/O interface of the Deed to the data store when the deed is created at resolve time within the declaring frame. The connective per is used to indicate that direct data follows. The field is the name of an associated attribute or action parameter to be initialized in the deed, and its value is the value to be assigned to that attribute or parameter. The Deed type determines whether its an attribute or action method parameter. Example:

```
do deed per setpoint ".mydeed.setpoint" gain ".mydeed.gain"
```

The option for allows injection of indirect data from a location in the store into the initialization of the I/O interface of the deed to the data store when the deed is created at resolve time within the declaring frame. The connective for is used to indicate that indirect data follows. Each field is the name of an associated attribute or action method parameter to be initialized in the deed, and each field's value is the pathname of a share to be assigned to that attribute or parameter. This adds one level of indirection to the case above. The Deed type determines whether its an attribute or action method parameter. Example:

```
do deed for setpoint gain in .init.mydeed
```

If the value of *setpoint* and *gain* in *.init.mydeed* are "*.mydeed.setpoint" and* "*.mydeed.gain"* then the two are equivalent.

The option with allows injection of direct data into the initialization of the deed instance when the deed is created at resolve time within the declaring frame. The connective with is used to indicate that direct data follows. The field is the name of an associated __init__ parameter and its value is the value to be assigned to that parameter. Example:

```
do deed with throttle 5
```

The option for allows injection of indirect data from a location in the store into the initialization of the Deed instance when the deed is created at resolve time within the declaring frame. The connective for is used to indicate that indirect data follows. Each field is the name of an associated __init__ method parameter, and each field's value is the value of that parameter. This adds one level of indirection to the case above. Example:

```
do deed from throttle in .init.mydeed
```

If the value of *throttle* in *.init.mydeed* is 5 then the two examples are equivalent.

7.7. go

Associated with each transition to another frame, is a set of pre-conditions that must be satisfied in order to make the transition to leave the current frame. An example transition pre-condition is as follows:

```
go homebase if fuelguage <= .10
```

The syntax for the go declaration follows:

```
go (frame, next, me) [if [not] need [and [not] need ...]]
```

The *go* declaration creates a new transition Action (*preact*) for the current Frame.

The native context is *precur* (prior to recur) and cannot be overridden.

A transition Action consists of transition target frame given by either its name *frame* or the word `next` or the word `me` and optionally the connective *if* followed by a list of need clauses which form the transition condition. The target Frame specifies the frame to be activated when the transition condition is satisfied. If no transition condition is provided then the transition is always taken.

When `next` is used it is replaced either by the Frame specified by the `next` declaration for the current Frame or if no `next` declaration is given, it defaults to the next frame to occur lexically in the FloScript.

When `me` is used it is replaced by the current Frame. This would make the transition a forced transition back to the same frame, thereby forcing an exit and enter.

The transition condition is a logical conjunction (*and*) of one or more need clauses. Each `and` connective creates a separate need clause. A need clause will be logically negated if it is preceded by the optional `not` connective. Each need clause may be of several forms which will be presented below in section 7.10 Need Syntax. Each need clause is implemented with a need Action. The transition will fail if the transition condition is not satisfied.

7.8. let

A given state may require certain resources or require that the vehicle be at a given position or orientation before the actions associated with the state can be executed. These are pre-conditions for entering the state. The rule is that, an outline may never be entered unless all the pre-conditions (Needs) are met. For example, a pre-condition for entering a frame, at the beginning of a mission, wherein the main propulsion motor is engaged would be to check that there is sufficient fuel to begin the mission, such as:

```
let me if fuelgauge >= .75
```

The syntax for the let declaration is as follows:

```
let [me] if [not] need [and [not] need ...]
```

The *let* declaration creates a new entry condition Action (*beact*)for the current frame.

The native context is *benter* (before enter) and cannot be overridden.

The entry condition is a logical conjunction (*and*) of one or more need clauses. Each `and` connective creates a separate need clause. A need clause will be logically negated if it is preceded by the optional `not` connective. Each need clause may be of several forms which will be presented below in section 7.10 Need Syntax. Each need clause is implemented with a need Action. The frame will not be entered if all the entry conditions are not satisfied.

7.9. aux

The aux declaration has two major forms. These are running an auxiliary framer and transitioning to a conditional auxiliary framer.

```
aux framer
```

Indicates that Framer named *framer* will be an auxiliary Framer of the current Frame.

```
aux framer if [not] need [and [not] need ...]
```

Creates a new conditional auxiliary Action (*preact*) for the current Frame.

The native context is *precur* (prior to recur) and cannot be overridden.

A conditional auxiliary Action consists of a target auxiliary given by its name *framer* and the connective *if* followed by a list of need clauses which form the condition. The target auxiliary specifies the auxiliary Framer to be activated when the condition is satisfied. Associated with a given mission are safety jackets the handle failure conditions that must be repaired before continuing with the mission. A conditional Auxiliary provides one way to implement the repair. For example,

```
aux getgpsfix if lost
```

The condition is a logical conjunction (*and*) of one or more need clauses. Each `and` connective creates a separate need clause. A need clause will be logically negated if it is preceded by the optional `not` connective. Each need clause may be of several forms which will be presented below in section 7.10 Need Syntax. Each need clause is implemented with a need Action. The auxiliary will not be activated if the condition is not satisfied.

When activated, a conditional auxiliary interrupts the execution of an subsequent preacts in the same frame and any subframes until the auxiliary is done. Then as long as the condition is not long satisfied normal operation of the frame and any subframes of the one holding the conditional auxiliary is resumed. A conditional auxiliary serves the same purpose as an interrupt service routine or exception routine but in the hierarchical action framework paradigm.

7.10. Need Syntax

The need Action syntax is presented below. Need Actions are only used for the conditions following the `if` connective in the *`let`*, *`go`*, and conditional *`aux`* declarations.

```
need:
   always
   done tasker
   status tasker is (readied, started, running, stopped, aborted)
   update [in frame] indirect
   change [in frame] indirect
   elapsed comparison goal [+- tolerance]
   recurred comparison goal [+- tolerance]
   state [comparison goal [+- tolerance]]

goal:
   value
   goal
   [(value, field) in] indirect

comparison:
    (==, !=, <, <=, >=, >)

state:
   [(value, field) in] indirect

tolerance:
    number  (the absolute value is used)
```

Creates an Action that evaluates to either True or False.

The `always` form always evaluates to True. This is useful for transitions that are part of a series of frames that must always be executed as fast as they can be stepped through. The associated Act references an instance named needAlways of the AlwaysNeed class.

Examples:

```
... if always
```

The `done` form evaluates if the tasker (including framers, servers, and loggers) named *`tasker`* is done. This is useful for transition to another frame once an auxiliary or slave is complete. The associated Act references an instance named needDone of the DoneNeed class. It applies to all Tasker scheduled types not just aux and slave.

Examples:

```
... if done gps
```

The `status` form evaluates if the status of the tasker (including framers, servers, and loggers) named *`tasker`* is one of `(readied, stopped, started, running, aborted)`. This is useful for controlling slave tasks. The associated Act references an instance named needStatus of the StatusNeed class.

Examples:

```
... if status gps is running
```

The `update` form evaluates if the *`indirect`* share has been updated within the *`frame`*. The default frame is the current lexical frame. Update compares a saved copy of the time stamp at frame entry with the time stamp of the share when the need evaluates . If the share time stamp when the need evaluates is >= the time stamp at frame entry then the need evaluates to True. This need inserts a special enact that saves the time stamp at frame entry.

Examples:

```
... if update .state.heading
... if update in frame start .state.heading
```

The `change` form evaluates if any fields in the data of the *`indirect`* share has been changed within the *`frame`*. The default frame is the current lexical frame. Change compares field by field, a copy of the share data made at frame entry with the data of the share when the need evaluates. If the share data when the need evaluates is != the share data at frame entry then the need evaluates to True. This need inserts a special enact that saves a copy of the data at frame entry.

Examples:

```
... if change .state.heading
... if change in frame start .state.heading
```

The `elapsed` form compares the elapsed time of the framer in seconds since entering the current outline with a time value given by *`goal`* using *`comparison`* and optional *`tolerance`*. If provided, the tolerance is ignored unless *`comparison`* is either == or `!=` . This form is not very useful as an entry condition since the elapsed time will always be zero, but is very useful as a transition pre-condition. The elapsed time is stored in `.framer.`*`name`*`.state.elapsed` where *`name`* is replaced with the name of the current Framer. The *`goal`* may be an explicit number, the word `goal`, the word `value` followed by an explicit number, or an indirect address. When *`goal`* is the word `goal`, the value is retrieved from `.framer.`*`name`*`.goal.elapsed` where *`name`* is replaced with the name of the current Framer. When *goal* is an indirect address, the implied goal variant of relative addressing is used. When the value of *`goal`* is given directly as a number, the associated Act references an instance named needDirect of the DirectNeed class. When the value of *`goal`* is provided by referencing a Share, the associated Act references an instance named needIndirect of the IndirectNeed class.

Examples:

```
... if elapsed == 0.0 +- 1.0
... if elapsed >= 15.0
```

The `recurred` form compares, the count of the number of times the framer has been iterated since entering the current outline, with the value provided by *`goal`*, using *`comparison`* and optional *`tolerance`*. If provided, the tolerance is ignored unless *`comparison`* is either == or != . This form is not very useful as an entry condition since the recurred count will always be zero, but is very useful as a transition pre-condition. The recurred count is zero the first time that outline entry and recur actions are evaluated. It is incremented before the transition conditions are checked. So the recurred count will be one after one iteration. A transition will always see a recurred count of one or more, and an entry condition will always see a recurred count of zero. So to execute the entry once and the recur actions twice do a transition if the `recurred` is >= 2. The recurred count is stored in `.framer.`*`name`*`.state.recurred` where *`name`* is replaced with the name of the current Framer. The *`goal`* may be an explicit number, the word `goal`, the word `value` followed by an explicit number, or an indirect address. When *`goal`* is the word `goal`, the value is retrieved from `.framer.`*`name`*`.goal.recurred` where *`name`* is replaced with the name of the current Framer. When *goal* is an indirect address, the implied goal variant of relative addressing is used. When the value of *`goal`* is given directly as a number, the associated Act references an instance named needDirect of the DirectNeed class. When the value of *`goal`* is provided by referencing a Share, the associated Act references an instance named needIndirect of the IndirectNeed class.

Examples:

```
... if recurred == 0.0
... if recurred >= 15
... if recurred > 3
```

The last form has several options. When no comparison clause is provided, it implicitly compares the value given by *`state`* to True, that is, `if` *`state`* is equivalent to `if` *`state`* `== true`. Likewise the negative form uses the `not` connective, that is, `if not` *`state`* is equivalent to `if` *`state`* `== false`. The implied state variant of relative addressing is used for the indirect address provided by *`state`*. The associated Act references an instance named needBoolean of the booleanNeed class.

Examples:

```
... if not leak
... if leak
```

When a comparison clause is provided, the value given by *state* is compared to the value given by *goal* using *comparison* and optional *tolerance*. If provided, the tolerance is ignored unless *comparison* is either == or != . The *goal* may be an explicit number, the word goal, the word value followed by an explicit number, or an indirect address. When *goal* is the word goal, the value is retrieved from .framer.*name*.goal.*path* where *name* is replaced with the name of the current Framer and *path* is the path portion of the indirect address for *state*. When *goal* is an indirect address, the implied goal variant of relative addressing is used. When the value of *goal* is given directly as a number, the associated Act references an instance named needDirect of the DirectNeed class. When the value of *goal* is provided by referencing a Share, the associated Act references an instance named needIndirect of the IndirectNeed class.

Examples:

```
... if depth < 5.0
... if depth >= goal
... if depth >= 10
... if speed == 2 +- 0.25
... if counter of frame > value 3
... if depth > 50
```

8. Other Declaration Verbs

8.1. print

```
print [token ...]

token:
   space or double quote delimited strings
```

The native context is *enter* but can be overridden. This declaration has no implied variants for its relative addressing.

Print creates an Action that prints to stdout a concatenation of the provided *values*. The associated Act references an instance named printer of the Printer class.

Examples:

```
print "Hello World"
>>> "Hello World"

print 1 2 3 4 5
>>> 1 2 3 4 5
```

8.2. load

```
load file
```

Loads the file name *file* and begins processing the FloScript declarations in the loaded file. Once completed, processing resumes on the next line after the load declaration. Load declarations may be nested so that mission scripts may be composed from many predefined script fragments.

8.3. house

```
house name
```

Creates new House of name *name*. All subsequent declarations apply to this House until the next `house` declaration is given. This also creates the shared Store for the House.

8.4. tasker

```
tasker name [part ...] [as kind [part ...]] [at pd] [rx h:p] [tx
h:p][be sd] [in or] [per data] [for source]


pd = period

h:p = host:port, (host:port, :port, host:, host, :)

sd = scheduled:
   (inactive, active, slave)

or = order:
   (front, mid, back)

px = prefix:
   filepath

data:
   direct

source:
   [(value, fields) in] indirect
```

Creates new tasker with name *name* modified by parts of name of kind *kind* modified by parts. The modifiers create camel case identifiers. If *kind* is not given then the kind is the kind of the named instance.

The tasker expects to be run every *period* seconds. If *period* is missing it is set to zero. A period of zero means run as often as possible.

The option `be` *`scheduled`* indicates the scheduling context of this tasker. If the `be` option is missing the default scheduling context is `inactive`. The scheduling context indicates to the Skedder how the server tasker is to be scheduled for execution as follows:

`inactive` means the tasker is initially stopped and waits for an explicit `bid` *`tasker`* `start` declaration after which the tasker is activated and the Skedder will automatically run the tasker until an explicit `bid` *`tasker`* `stop` or `done` declaration is encountered.

`active` means the tasker is initially started by the Skedder and will be run automatically by the tasker until an explicit `bid` *`tasker`* `stop` or `done` declaration is encountered. Once the tasker as been stopped it is now inactive and can be reactivated if a subsequent `bit` *`tasker`* `start` declaration is encountered.

`slave` means the tasker is not executed by the Skedder and can only be run as the slave of another tasker in response to explicit `tell` *`tasker`* `start`, `tell` *`tasker`* `run`, `tell` *`tasker`* `stop`, or `done` declarations. The tasker is never run automatically but only in response to a `tell` *`tasker`* `run` declaration.

The option `in` *`order`* indicates the scheduling order of this tasker relative to other tasks tasked by the tasker. This option is ignored if the `be` option is not `active` or `inactive`. If the `in` option is missing the default place is `mid`. This option is useful when there is more than one house. This allows tasks from later houses to be executed before tasks of earlier houses and vice versa.

```
./logs/uuv_remus_20090518_172931/
```

The option `per` allows injection of direct data into the initialization of the tasker. The connective `per` is used to indicate that direct data follows. Example:

```
tasker mytask per period 1
```

The option `for` allows injection of indirect data from a location in the store into the initialization of the tasker. The connective `for` is used to indicate that indirect data follows. Example:

```
tasker mytask for value in .init.period
```

```
./logs/uuv_remus_20090518_172931/
```

Example:

```
tasker foo bar as trouble stuff at 0.0 be active
```

8.5. server

```
server name [part ...] [as kind [part ...]] [at pd] [rx h:p] [tx
h:p][be sd] [in or] [to px] [per data] [for source]

pd = period

h:p = host:port, (host:port, :port, host:, host, :)

sd = scheduled:
   (inactive, active, slave)

or = order:
   (front, mid, back)

px = prefix:
   filepath

data:
   direct

source:
   [(value, fields) in] indirect
```

Creates new server tasker with name *name* modified by parts of kind *kind* modified by parts. The modifiers create camel case identifiers. If *kind* is not given then the kind is the kind of the named instance.

The server expects to be run every *period* seconds. If *period* is missing it is set to zero. A period of zero means run as often as possible.

The `tx and rx` options provide IPv4 hosts in the form "xxx.xxx.xxx.xxx" and the port as a number. Each type of server has a default values for the `rx` and `tx` options. The `rx` option is the host address and port of the the server is listening on. The server receives IP messages from this address. An empty host address indicates receive from any IP interface on the host running this HAF. The *tx* option provides the host address and port that the server sends IP messages to.

The option `be` *scheduled* indicates the scheduling context of this tasker. If the `be` option is missing the default scheduling context is `inactive`. The scheduling context indicates to the Skedder how the server tasker is to be scheduled for execution as follows:

`inactive` means the tasker is initially stopped and waits for an explicit `bid` *tasker* `start` declaration after which the tasker is activated and the Skedder will automatically run the tasker until an explicit `bid` *tasker* `stop` or `done` declaration is encountered.

`active` means the tasker is initially started by the Skedder and will be run automatically by the tasker until an explicit `bid` *tasker* `stop` or `done` declaration is encountered. Once the tasker as been stopped it is now inactive and can be reactivated if a subsequent `bit` *tasker* `start` declaration is encountered.

`slave` means the tasker is not executed by the Skedder and can only be run as the slave of another tasker in response to explicit `tell` *`tasker`* `start`, `tell` *`tasker`* `run`, `tell` *`tasker`* `stop`, or `done` declarations. The tasker is never run automatically but only in response to a `tell` *`tasker`* `run` declaration.

The option `in` *`order`* indicates the scheduling order of this tasker relative to other tasks tasked by the tasker. This option is ignored if the `be` option is not `active` or `inactive`. If the `in` option is missing the default place is `mid`. This option is useful when there is more than one house. This allows tasks from later houses to be executed before tasks of earlier houses and vice versa.

The option `to prefix` is used to generated the log directory for this server. All the files generated by this server are stored in this directory. The prefix is the directory path. The full name of the directory is the prefix concatenated with the base name. The base name of the directory is composed of the house name, the server name and the date and time from the system clock that the server started running separated by underscores. If the `to prefix` option is missing then `'./'` is used for the prefix (the current working directory). For example, given prefix = `./logs/`, house name = `uuv`, server name = `remus`, a date of `2009/05/18` and a time of `17:29:31`, the following log directory name will be created:

```
./logs/uuv_remus_20090518_172931/
```

The option `per` allows injection of direct data into the initialization of the tasker. The connective `per` is used to indicate that direct data follows. Example:

```
server myserver per period 1.25
```

The option `for` allows injection of indirect data from a location in the store into the initialization of the tasker. The connective `for` is used to indicate that indirect data follows. Example:

```
server myserver for period in .init.period
```

Example:

```
server autopilot as recon remus at 0.0 be active rx :23456 rx
192.168.1.77:23456
```

8.6. logger

```
logger name [at pd] [to px] [be sd] [in or] [flush il]

pd = period

px = prefix
```

```
sd = scheduled:
   (inactive, active, slave)

or = order:
   (front, mid, back)

il = interval
```

Creates new data logger tasker of name *name*. The `logger` declaration applies to all `log` declaration until the next `logger` declaration appears.

The option `at` *`period`* indicates the desired time in seconds between automatic evaluations of the logger by the tasker. Period is ignored if the scheduled is slave. If `at` *`period`* is missing then *`period`* is set to zero. A period of zero means run as often as possible, that is, every time the tasker runs.

The option `to prefix` is used to generated the log directory for this logger. All the logs run by this logger are stored in this directory. The prefix is the directory path. The full name of the directory is the prefix concatenated with the base name. The base name of the directory is composed of the house name, the logger name and the date and time from the system clock that the logger started running separated by underscores. If the `to prefix` option is missing then `'./'` is used for the prefix (the current working directory). For example, given prefix = `./logs/`, house name = `uuv`, logger name = `main`, a date of `2009/05/18` and a time of `17:29:31`, the following log directory name will be created:

```
./logs/uuv_main_20090518_172931/
```

The option `be` *`scheduled`* indicates the scheduling context of this tasker. If the `be` option is missing the default scheduling context is `inactive`. The scheduling context indicates to the Skedder how the server tasker is to be scheduled for execution as follows:

`inactive` means the tasker is initially stopped and waits for an explicit `bid` *`tasker`* `start` declaration after which the tasker is activated and the Skedder will automatically run the tasker until an explicit `bid` *`tasker`* `stop` or `done` declaration is encountered.

`active` means the tasker is initially started by the Skedder and will be run automatically by the tasker until an explicit `bid` *`tasker`* `stop` or `done` declaration is encountered. Once the tasker as been stopped it is now inactive and can be reactivated if a subsequent `bid` *`tasker`* `start` declaration is encountered.

`slave` means the tasker is not executed by the Skedder and can only be run as the slave of another tasker in response to explicit `tell` *`tasker`* `start`, `tell` *`tasker`* `run`, `tell` *`tasker`* `stop`, or `done` declarations. The tasker is never run automatically but only in response to a `tell` *`tasker`* `run` declaration.

The option `in` *`order`* indicates the scheduling order of this tasker relative to other tasks tasked by the tasker. This option is ignored if the `be` option is not `active` or `inactive`. If the `in` option is

missing the default place is `mid`. This option is useful when there is more than one house. This allows tasks from later houses to be executed before tasks of earlier houses and vice versa.

The option `flush` *`interval`* indicates time in seconds between flushes to disk of the log files. The minimum flush interval is 1 second. The default is 30 seconds.

Example:

```
logger main at 0.0 to ./logs/ be active
```

8.7. log

```
log name [as (text, binary)] [to file][on rule]

rule:
(once, never, always, update, change, lifo, fifo)
```

Creates new data Log of name *`name`* for saving share values to file storage.

The `log` declaration applies to all `loggee` declarations until the next `log` declaration appears.

The data format option `as (text, binary)` indicates what data format is used in the log file. The log file has a text header that documents the field names. The header is always text even if the data format is binary. Currently only text is supported. If the data format option is missing then text is used.

The option, `to` *`file`* specifies the base name of the log file where the log data will be written. The loge file will be placed in the log directory of this logs's logger. If the `to` *`file`* option is missing then the log name is used as the file name.

The option, `on` *`rule`* specifies the condition that must be satisfied for data to be logged. The rules are `(once, never, always, update, change, lifo, fifo).`

`once` means log once at the start, useful for recording unchanging parameters.

`never` means don't log until explicitly declarationed by an action.

`always` means log every time the log is evaluated by its logger.

`update` means log anytime the time stamp on the share(s) is newer than the last time data was saved. It also logs the initial values at start.

`change` means log if the any of the values in the shares have changed since the last time data was saved. It also logs the initial values at start.

`lifo` means log all items in the sequence in lifo order from the first field of the first loggee . This is a special rule that is meant to used when the value of the loggee share is a sequence or mapping. This rule will remove all the elements from the sequence each time it logs. So this should only be applied

to share values that are not consumed by other behaviors. This rule makes it possible to log python lists or dicts.

`fifo` means log all items in the sequence in lifo order from the first field of the first loggee . This is a special rule that is meant to used when the value of the loggee share is a sequence or mapping. This rule will remove all the elements from the sequence each time it logs. So this should only be applied to share values that are not consumed by other behaviors. This rule makes it possible to log python lists or dicts.

Example:

```
log state on update
```

8.8. loggee

`loggee` *`tag path [tag path ...]`*

Indicates which shares to include in the log file. Each loggee is denoted by a *`tag`*. The *`tag`* is placed in the header of the log and is a convenient way of keeping track of the Share. When a Share has multiple fields, the log heading uses dotted notation *tag.field* to denote each field. The *`path`* is the name in doted notation of the share. Multiple `loggee` declarations may be given for a single log. All the loggees are concatenated in the associated log.

Example:

```
loggee heading state.heading depth state.depth speed state.speed
```

8.9. framer

`framer` *`name`* [`at` *`period`*] [`first` *`frame`*][`be` *`scheduled`*] [`in` *`order`*]

`scheduled:`
```
   (inactive, active, aux, slave)
```

`order`:
```
   (front, mid, back)
```

Creates new Framer with unique name *`name`*. All subsequent Frame and Action declarations apply to this framework until the next `framer` declaration is given.

The option `at` *`period`* indicates the desired time in seconds between automatic evaluations of the framework by the tasker. Period is ignored if the scheduling context is aux or slave. If `at` *`period`* is missing then *`period`* is set to zero. A period of zero means run as often as possible, that is, every time the tasker runs.

The option `first` *`frame`* indicates that the frame named *frame* is to the the beginning frame for the framework. If the start option is missing then the starting frame will be indicated by a `first` declaration or if no `first` declaration the first frame encountered lexically after the Framer declaration.

The option `be` *`scheduled`* indicates the scheduling context of this tasker. If the `be` option is missing the default scheduling context is `inactive`. The scheduling context indicates to the Skedder how the tasker is to be scheduled for execution as follows:

`inactive` means the tasker is initially stopped and waits for an explicit `ask` *`tasker`* `start` declaration after which the tasker is activated and the Skedder will automatically run the tasker until an explicit `bid` *`tasker`* `stop` or `done` declaration is encountered.

`active` means the tasker is initially started by the Skedder and will be run automatically by the tasker until an explicit `bid` *`tasker`* `stop` or `done` declaration is encountered. Once the tasker as been stopped it is now inactive and can be reactivated if a subsequent `bid` *`tasker`* `start` declaration is encountered.

`aux` means the tasker is not executed by the Skedder and can only be run as the auxiliary of another framer.

`slave` means the tasker is not executed by the Skedder and can only be run as the slave of another framework in response to explicit `tell` *`tasker`* `start`, `tell` *`tasker`* `run`, `tell` *`tasker`* `stop`, or `done` declarations. The tasker is never run automatically but only in response to a `tell` *`tasker`* `run` declaration.

The option `in` *`order`* indicates the scheduling order of this tasker relative to other tasks tasked by the tasker. This option is ignored if the `be` option is not `active` or `inactive`. If the `in` option is missing the default place is `mid`. This option is useful when there is more than one house. This allows tasks from later houses to be executed before tasks of earlier houses and vice versa.

Example:

```
framer mission at 0.125 be active first leg
```

8.10. first

`first` *`frame`*

Indicates that the frame named *frame* is to the the starting Frame for the Framer. Overrides any previous `start` declaration or the `start` option in the `framer` declaration.

Example:

```
first leg
```

8.11. frame

```
frame name [in over]
```

Creates new Frame with unique name *name* for the current Framer. Option *over* specifies that frame named *over* is to be the over frame of this frame. All subsequent action declarations apply to this frame until the next frame declaration is given. Since Frame names are only unique per framer, the same frame name can be reused in multiple Framers.

8.12. over

```
over frame
```

Indicates that Frame named *frame* is to be the over Frame of the current Frame. Overrides any previous `over` declaration or the `over` option in the frame declaration.

8.13. under

```
under frame
```

Indicates that Frame named *frame* is to be the primary under Frame of the current Frame. Overrides any previous `under` declaration or default.

8.14. next

```
next frame
next
```

Indicates that Frame named *frame* is to be the default next Frame for any transitions from the current frame that specify `next` as the target. Overrides any previous `next` declaration or default. If no next declaration is provided for the current Frame then the next lexically appearing Frame in the FloScript will default as the next Frame which is the default.

8.15. native

```
native
```

Reverts the current context to native for applicable actions until a new `enter, recur, precur, exit, rexit, or renter` declaration is encountered or until a new frame starts.

8.16. enter

`enter`

Sets the current context to enter for applicable actions until a new `native, enter, recur, precur, exit, rexit, or renter` declaration is encountered or until a new frame starts.

8.17. recur

`recur`

Sets the current context to recur for applicable actions until a new `native, enter, recur, precur, exit, rexit, or renter` declaration is encountered or until a new frame starts.

8.18. precur

`precur`

Sets the current context to recur for applicable actions until a new `native, enter, recur, precur, exit, rexit, or renter` declaration is encountered or until a new frame starts.

8.19. exit

`exit`

Sets the current context to exit for applicable actions until a new `native, enter, recur, precur, exit, rexit, or renter` declaration is encountered or until a new frame starts.

8.20. rexit

`rexit`

Sets the current context to exit for applicable actions until a new `native, enter, recur, precur, exit, rexit, or renter` declaration is encountered or until a new frame starts.

8.21. renter

`renter`

Sets the current context to exit for applicable actions until a new `native, enter, recur, precur, exit, rexit, or renter` declaration is encountered or until a new frame starts.

8.22. done

```
done
```

The native context is *enter* but may be overridden. Immediately sets the current Framer's done attribute to True so that the some Action can check for its completion. May only be used in a Frame of an auxiliary or slave Framer. Typical usage is to use the done declaration in the last frame of the auxiliary or slave to indicate to Main frame that auxiliary or slave has completed

8.23. repeat

```
repeat value
```

Creates an implicit transition action with target `next` that succeeds when the recurred count in the current outline reaches or exceeds *`value`*. Shortcut for,

```
go next if recurred >= value
```

8.24. timeout

```
timeout value
```

Creates an implicit transition action with target `next` that succeeds when the elapsed time in the current outline reaches or exceeds *`value`*. Shortcut for,

```
go next if elapsed >= value
```

8.25. bid

```
bid control tasker [tasker ...]

control:
   (start, stop, run)

tasker:
   (name, me, all)
```

The native context is *enter* but can be overridden. Creates a new Want action that sends the *`control`* to the list of *tasks* given by the Task's name *name*. The `me` form sends the *`control`* to the current Framer. The all form send the *`control`* to all the taskable tasks in the current house. Control occurs by setting the Tasker.desire attribute. The tasker passes in the value of this attribute when running the Task's generator. Thus bid could be used to request that the tasker start a tasker running or stop a tasker from running. By default a tasker will set its desire to run so once running it will continue running unless something tells it to stop. The bid declaration is primarily intended for

use by one framer to stop other tasks and framers at the end of a mission. It cannot be used on auxiliary framers or slave framers.

Example:

```
bid stop me
bid stop all
bid stop gps
bid stop gps mission
```

8.26. ready

`ready` *`tasker`*

The native context is *benter* but can be overridden. Creates new fiat entry Action to send a *ready* control to the Tasker named *`tasker`*. This declaration only applies to slave Tasks. If the tasker if a slave framer then it will check if the framer can be entered which is useful as a benter action.

8.27. start

`start` *`tasker`*

The native context is *enter* but can be overridden. Creates new fiat entry Action to send a *start* control to the Tasker named *`tasker`*. This declaration only applies to slave Tasks. A separate `run` *`tasker`* is needed for each iteration of the tasker.

8.28. stop

`stop` *`tasker`*

The native context is *exit* but can be overridden. Creates new fiat entry Action to send a *stop* control to the Tasker named *`tasker`*. This declaration only applies to slave Tasks.

8.29. run

`run` *`tasker`*

The native context is *recur* but can be overridden. Creates new fiat entry Action to send a *run* control to the Tasker named *`tasker`*. This declaration only applies to slave Tasks. A separate *run*

declaration is needed for each iteration of the slave tasker. Changing the context to recur will send a *run* declaration for each recur the the frame.

8.30. abort

`abort` *`tasker`*

The native context is *enter* but can be overridden. Creates new fiat entry Action to send an *abort* control to the Tasker named *`tasker`*. This declaration only applies to slave Tasks.

9. Unfinished Declaration Verbs

A few more declarations have been designed but not yet added to the FloScript.

10. Example FloScripts

The important insight to observe while going through the examples is that as each mission gets more complicated all the associated configuration and high level mission logic is still self-contained in FloScript. The only time one has to go outside FloScript is to write the code for the behaviors (deeds). Scheduling the behaviors, publishing and subscribing to the Store, logging Shares, making transitions, forming hierarchical state machine logic, etc is all transparently explicated in FloScript.

10.1. Simple Non Hierarchical Mission

This example is a very simple mission that commands the vehicle to follow four different headings, forming a box. This mission depends on several deeds for simulating and controlling the vehicle. The uuv motion simulator updates the heading depth and speed states of the vehicle in the .state.heading, .state.depth and .state.speed Shares respectively. The depth pid controller updates the pitch set-point to achieve the depth set-point in .goal.depth which is in turn used by the pitch controller to update the stern plane position used by the simulator. The heading pid controller updates the rudder position used by the simulator to achieve the to achieve the heading set-point in .goal.heading. The speed controller updates the propellor rpm used by the simulator to achieve the speed set-point in .goal.speed. The other details of the deeds are immaterial to understanding the associated FloScript.

```
#example mission box1.flo

house box1

framer vehiclesim be active first vehicle_run

frame vehicle_run
   do simulator motion uuv

framer mission be active first northleg

frame northleg
   set elapsed to 20.0
   set heading to 0.0
   set depth to 5.0
   set speed to 2.5
   go next if elapsed >= goal

frame eastleg
   set heading to 90.0
   go next if elapsed >= goal

frame southleg
   set heading to 180.0
   go next if elapsed >= goal

frame westleg
   set heading to 270.0
   go next if elapsed >= goal
```

```
frame mission_stop
   bid stop vehiclesim
   bid stop autopilot
   bid stop me

framer autopilot be active first autopilot_run

frame autopilot_run
   do controller pid speed
   do controller pid heading
   do controller pid depth
   do controller pid pitch
```

10.2. Simple Hierarchical Mission

This example extends the previous example to add a hierarchical frame to prevent the depth from exceeding a maximum. If the maximum is exceeded the mission framer goes to the frame abort to bring the vehicle back to the surface. The maximum depth value is initialized in the Share .max.depth.

This example also runs a logger with two logs. One to record the heading, depth, and speed of the vehicle and the other to record the commanded heading, depth, and speed set-points.

```
#example mission box2.flo
# with max depth protection

house box2

init max.depth to 50.0

framer vehiclesim be active first vehicle_run

frame vehicle_run
   do simulator motion uuv

framer mission be active first northleg

frame depthmax
   go abort if depth >= .max.depth

frame northleg in depthmax
   set elapsed to 20.0
   set heading to 0.0
   set depth to 5.0
   set speed to 2.5
   go next if elapsed >= goal
```

```
frame eastleg in depthmax
   set heading to 9.0
   go next if elapsed >= goal

frame southleg in depthmax
   set heading to 180.0
   go next if elapsed >= goal

frame westleg in depthmax
   set heading to 270.0
   go next if elapsed >= goal

frame mission_stop
   bid stop vehiclesim
   bid stop autopilot
   bid stop logger
   bid stop me

frame abort
   set depth to 0.0
   set speed to 2.5
   go mission_stop if depth == 0.0 +- 0.25

framer autopilot be active first autopilot_run

frame autopilot_run
   do controller pid speed
   do controller pid heading
   do controller pid depth
   do controller pid pitch


logger logger to ../log/
log state  on update
loggee position state.position heading state.heading depth
state.depth speed state.speed

log goal on update
loggee heading goal.heading depth goal.depth speed goal.speed
```

10.3. Simple Hierarchical Mission with Auxiliary

This example extends the previous example to add an auxiliary Framer to bring the vehicle to the surface to get a gps fix at the end of each leg.

```
#example mission box3.flo
#with auxiliary for gps fix
```

```
house box3

init max.depth to 50.0

framer vehiclesim be active first vehicle_run

frame vehicle_run
   do simulator motion uuv

framer mission be active first northleg

frame depthmax
   go abort if depth >= .max.depth

frame northleg in depthmax
   set elapsed to 20.0
   set heading to 0.0
   set depth to 5.0
   set speed to 2.5
   go next if elapsed >= goal

frame gpsfixnorth
   aux gps
   go next if done gps

frame eastleg in depthmax
   set elapsed to 20.0
   set heading to 90.0
   set depth to 5.0
   set speed to 2.5
   go next if elapsed >= goal

frame gpxfixeast
   aux gps
   go next if done gps

frame southleg in depthmax
   set elapsed to 20.0
   set heading to 180.0
   set depth to 5.0
   set speed to 2.5
   go next if elapsed >= goal

frame gpxfixsouth
   aux gps
   go next if done gps
```

```
frame westleg in depthmax
   set elapsed to 20.0
   set heading to 270.0
   set depth to 5.0
   set speed to 2.5
   go next if elapsed >= goal

frame gpxfixwest
   aux gps
   go next if done gps

frame mission_stop
   bid stop vehiclesim
   bid stop autopilot
   bid stop logger
   bid stop me

frame abort
   set depth to 0.0
   set speed to 2.5
   go mission_stop if depth == 0.0 +- 0.25

framer autopilot be active first autopilot_run

frame autopilot_run
   do controller pid speed
   do controller pid heading
   do controller pid depth
   do controller pid pitch

logger logger to ../log/
log state on update
loggee position state.position heading state.heading depth
state.depth speed state.speed

log goal on update
loggee heading goal.heading depth goal.depth speed goal.speed

framer gps be aux first gps1

frame gps1
   set depth to 0.0
   go next if depth == 0.0 +- 0.25

frame gps2
   set speed to 0.0
   go next if elapsed >= 10.0
```

```
frame gps3
   done
```

10.4. Re-factored Mission

This example re-factors the previous example to use the `inc` declaration to perform the box by repeating the same frame four times but with a different heading. The heading reference is inited in box.heading. One could just code a behavior (deed) that generates the sequence of headings and run that behavior. But this example shows how basic repeating sequences can be performed in FloScript itself for quick testing using the minimalist set of FloScript declarations. Also the multiple bid stop declarations to stop the framers have been replaced with one `bid stop all` declaration.

```
#example mission box4.flo
# with refactored leg iteration using counter

house box4

init max.depth to 50.0

init boxer.heading to 0.0
init boxer.turn to 90.0
init boxer.leg to 0.0
init boxer.nlegs to 4.0

framer vehiclesim be active first vehicle_run

frame vehicle_run
   do simulator motion uuv

framer mission be active first leg

frame depthmax
   go abort if depth >= .max.depth

frame leg in depthmax
   set elapsed to 20.0
   set heading from boxer.heading
   set depth to 5.0
   set speed to 2.5
   go next if elapsed >= goal
   go mission_stop if .boxer.leg >= .boxer.nlegs

frame gpsfix
   aux gps
   go next if done gps

frame turn in depthmax
   inc boxer.heading from boxer.turn
```

```
   inc boxer.leg by 1.0
   go leg

frame abort
   set depth to 0.0
   set speed to 2.5
   go mission_stop if depth == 0.0 +- 0.25

frame mission_stop
   bid stop all

framer autopilot be active first autopilot_run

frame autopilot_run
   do controller pid speed with fake 1
   do controller pid heading
   do controller pid depth from value in .max.depth
   do controller pid pitch

logger logger to ../log/
log state  on update
loggee position state.position heading state.heading depth
state.depth speed state.speed

log goal on update
loggee heading goal.heading depth goal.depth speed goal.speed

framer gps be aux first gps1

frame gps1
   set depth to 0.0
   go next if depth == 0.0 +- 0.25

frame gps2
   set speed to 0.0
   go next if elapsed >= 10.0

frame gps3
   done
```

10.5. Mission using load declaration

This example again re-factors the previous example to use the load declaration to load the gps auxiliary from a separate file. It also gets rid of Frame "turn" by moving its actions to exit actions in Frame gpsfix.

```
# example mission box5.flo
# refactored box mission with load
```

```
# uses exit context to increment counter

house box5

init max.depth to 50.0

init boxer.heading to 0.0
init boxer.turn to 90.0
init boxer.leg to 0.0
init boxer.nlegs to 4.0

framer vehiclesim be active first vehicle_run

frame vehicle_run
   do simulator motion uuv

framer mission be active first leg

frame depthmax
   go abort if depth >= .max.depth

frame leg in depthmax
   set elapsed to 20.0
   set heading from boxer.heading
   set depth to 5.0
   set speed to 2.5
   go gpsfix if elapsed >= goal
   go mission_stop if .boxer.leg >= .boxer.nlegs

frame gpsfix
   aux gps
   go leg if done gps
   exit
      inc boxer.heading from boxer.turn
      inc boxer.leg by 1.0

frame abort
   set depth to 0.0
   set speed to 2.5
   go mission_stop if depth == 0.0 +- 0.25

frame mission_stop
   bid stop all

framer autopilot be active first autopilot_run

frame autopilot_run
   do controller pid speed
```

```
   do controller pid heading
   do controller pid depth
   do controller pid pitch

logger logger to ../log/
log state   on update
loggee position state.position heading state.heading depth
state.depth speed state.speed

log goal   on update
loggee heading goal.heading depth goal.depth speed goal.speed

load ../plan/gps.flo
```

10.6. Loaded file

The gps auxiliary is stored in the file `gps.flo` with the following contents:

```
#example mission gps.flo
#gps fix auxiliary for load declaration

framer gps be aux first gps1

frame gps1
   set depth to 0.0
   go next if depth == 0.0 +- 0.25

frame gps2
   set speed to 0.0
   go next if elapsed >= 10.0

frame gps3
   done
```

10.7. Mission using conditional auxiliary to repair depth overage

This example adds a conditional auxiliary named "shallow" to bring the vehicle near the surface if it exceeds the max depth. This is an example of using hierarchical frames to provide a safety jacket. It also uses a beact.

```
# example mission box6.flo
# uses conditional auxiliary

house box6

init limit.depth.max to 50.0
init limit.depth.shallow to 10.0
init limit.depth.cruise to 5.0
```

```
init boxer.heading to 0.0
init boxer.turn to 90.0
init boxer.leg to 0.0
init boxer.nlegs to 4.0

framer vehiclesim be active first vehicle_run

frame vehicle_run
   do simulator motion uuv

framer mission be active first leg

frame depthmax
   go abort if depth >= .limit.depth.max
   aux shallow if depth >= .limit.depth.shallow # until aux done

frame leg in depthmax
   set elapsed to 60.0
   set heading from boxer.heading
   set depth to 15.0
   set speed to 2.5
   go gpsfix if elapsed >= goal
   go mission_stop if .boxer.leg >= .boxer.nlegs

frame gpsfix
   aux gps
   go leg if done gps
   exit
      inc boxer.heading from boxer.turn
      inc boxer.leg by 1.0

frame abort
   set depth to 0.0
   set speed to 2.5
   go mission_stop if depth == 0.0 +- 0.25

frame mission_stop
   bid stop all

framer autopilot be active first autopilot_run

frame autopilot_run
   do controller pid speed
   do controller pid heading
   do controller pid depth
   do controller pid pitch
```

```
framer shallow be aux first shallow1 # conditional auxiliary

frame shallow1
   set depth to 0.0
   set speed to 2.5
   go next if depth <= .limit.depth.cruise #repaired

frame shallow2 #so complete
   let me if depth <= .limit.depth.cruise # dumb beact
   done

logger logger to ../log/
log state   on update
loggee position state.position heading state.heading depth
state.depth speed state.speed

log goal   on update
loggee heading goal.heading depth goal.depth speed goal.speed
duration goal.duration

load ../plan/gps.flo
```

10.7.1. External gps.flo file

```
#example mission gps.flo
#gps fix auxiliary for load declaration

framer gps be aux first gps1

frame gps1
   set depth to 0.0
   go next if depth == 0.0 +- 0.25

frame gps2
   set speed to 0.0
   go next if elapsed >= 10.0

frame gps3
   done
```

10.8. Miscellaneos declarations

This example shows an example of controlling a slave auxiliary framer as well as several other declarations.

```
# meta.flo
# bootstrap house and framer that builds rest of houses and framers
```

```
# This is a test of the line continuation \
# how does this work

house meta

init meta.name to value "test"
init meta.spot to when "now" where "here"
init meta.name to value "test2"

framer testmeta be active first testmeta

frame testmeta
   print "Hello World"
   set elapsed to 1.0
   go next if elapsed >= goal

frame testtimeout
   print "timeout"
   timeout 1.0

frame testrecurred
   print "recurred"
   set recurred to 4
   recur
      print "recurred again"
   go next if recurred >= goal

frame testrepeat
   print "repeat"
   recur
      print "repeated again"
   repeat 2

frame doslave
   ready testslave #benter
   start testslave # enter
   run testslave # recur
   go next if done testslave
   stop testslave # exit

frame abortslave
   print abortslave
   abort testslave
   go next if status testslave is aborted

frame finish
   bid stop all
```

```
logger logger to ../log/
log meta on once
loggee name meta.name version meta.version period meta.period


framer testslave be slave first slave1 # slave framer

frame slave1
   print slave1
   go next

frame slave2
   print slave2
   repeat 3
   recur
      print slave2 again

frame slave3 #so complete
   print slave3
   done

server server one as server to ../log/ rx :55551 with stuff 5
server server two as server to ../log/ \
   rx :55552 \
   from value in .meta.name # trailing comment
server server to ../log/ # trailing comment

tasker tasker one as tasker  with stuff 5
tasker tasker two as tasker  from value in .meta.name
tasker tasker
```

11. FloScript Design Patterns

This section documents some simple design patterns for doing useful things with FloScript.

11.1. Repeating an Auxiliary

This patterns shows how to use forced entry/exit of a frame to run and count the repetitions of an auxiliary in one frame.

```
#example mission counter1.flo
# Design pattern: Counting iterations of aux framer
# uses forced enter exit to inc a counter and
# run aux all in one frame

house uuv

framer mission be active first leg
```

```
frame setup
   set .leg to 0
   set .nlegs to 3

frame leg in setup
   aux myaux
   go me if done myaux  #forced exit entry
   go finish if .leg >= .nlegs
   exit
      inc .leg by 1

frame finish
   bid stop uuvlogger
   bid stop me

logger uuvlogger to ./logs/
log leg on update
loggee leg leg

framer myaux be aux first myaux1

frame myaux1
   go next if elapsed >= 1.0

frame myaux2
   timeout 1.0

frame myaux3
   done
```

12. Supporting Behaviors

This section documents some of the behaviors that have been developed.

This section To be completed.