An Overview of MOOS-IvP and a Users Guide to the IvP Helm - Release 13.2

Michael R. Benjamin[1], Henrik Schmidt[1], Paul Newman[2], John J. Leonard[1]

[1]Department Mechanical Engineering
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge MA

[2]Department of Engineering Science
University of Oxford, Oxford England

February 4th, 2013 - Release 13.2

Abstract

This document describes the IvP Helm - an Open Source behavior-based autonomy application for unmanned vehicles. IvP is short for interval programming - a technique for representing and solving multi-objective optimizations problems. Behaviors in the IvP Helm are reconciled using multi-objective optimization when in competition with each other for influence of the vehicle. The IvP Helm is written as a MOOS application where MOOS is a set of Open Source publish-subscribe autonomy middleware tools. This document describes the configuration and use of the IvP Helm, provides examples of simple missions and information on how to download and build the software from the MOOS-IvP server at www.moos-ivp.org.

This work is the product of a multi-year collaboration between the Department of Mechanical Engineering and the Computer Science and Artificial Intelligence Laboratory (CSAIL) at the Massachusetts Institute of Technology in Cambridge Massachusetts, and the Oxford University Mobile Robotics Group.

Points of contact for collaborators:

Dr. Michael R. Benjamin
Department of Mechanical Engineering
Computer Science and Artificial Intelligence Laboratory
Massachusetts Intitute of Technology
mikerb@csail.mit.edu

Dr. Paul Newman
Department of Engineering Science
University of Oxford
pnewman@robots.ox.ac.uk

Prof. Henrik Schmidt
Department of Mechanical Engineering
Massachusetts Intitute of Technology
henrik@mit.edu

Prof. John J. Leonard
Department of Mechanical Engineering
Computer Science and Artificial Intelligence Laboratory
Massachusetts Intitute of Technology
jleonard@csail.mit.edu

Other collaborators have contributed greatly to the development and testing of software and ideas within, notably - Joseph Curcio, Toby Schneider, Stephanie Kemna, Arjan Vermeij, Don Eickstedt, Andrew Patrikilakis, Arjuna Balasuriya, David Battle, Christian Convey, Chris Gagner, Andrew Shafer, and Kevin Cockrell.

Sponsorship, and public release information:

This work is sponsored by Dr. Behzad Kamgar-Parsi and Dr. Don Wagner of the Office of Naval Research (ONR), Code 311. Further support for testing and coursework development sponsored by Battelle, Dr. Robert Carnes.

Contents

1 Overview

1.1 Purpose and Scope of this Document

The purpose of this document is to provide an overview of the IvP Helm in terms of design considerations, architecture and usage. This document contains references to example missions distributed with the MOOS-IvP software bundle at `www.moos-ivp.org`. The example and material herein should serve as a "getting-started" guide as well as users manual for users looking to go beyond simple autonomy missions. *This document represents release 13.2 and is still a work in progress. It is still considered to be in draft form and has know omissions. The reader is encouraged to email the authors feedback and look for later versions.*

1.2 Brief Background of MOOS-IvP

MOOS was written by Paul Newman in 2001 to support operations with autonomous marine vehicles in the MIT Ocean Engineering and the MIT Sea Grant programs. At the time Newman was a post-doc working with John Leonard and has since joined the faculty of the Mobile Robotics Group at Oxford University. MOOS continues to be developed and maintained by Newman at Oxford and the most current version can be found at his web site. The MOOS software available in the MOOS-IvP project includes a snapshot of the MOOS code distributed from Oxford. The IvP Helm was developed in 2004 for autonomous control on unmanned marine surface craft, and later underwater platforms. It was written by Mike Benjamin as a post-doc working with John Leonard, and as a research scientist for the Naval Undersea Warfare Center in Newport Rhode Island. The IvP Helm is a single MOOS process that uses multi-objective optimization to implement behavior coordination.

Acronyms

MOOS stands for "Mission Oriented Operating Suite" and its original use was for the Bluefin Odyssey III vehicle owned by MIT. IvP stands for "Interval Programming" which is a mathematical programming model for multi-objective optimization. In the IvP model each objective function is a piecewise linear construct where each piece is an *interval* in N-Space. The IvP model and algorithms are included in the IvP Helm software as the method for representing and reconciling the output of helm behaviors. The term interval programming was inspired by the mathematical programming models of linear programming (LP) and integer programming (IP). The pseudo-acronym IvP was chosen simply in this spirit and to avoid acronym clashing.

1.3 Sponsors of MOOS-IvP

Original development of MOOS and IvP were more or less infrastructure by-products of other sponsored research in (mostly marine) robotics. Those sponsors were primarily The Office of Naval Research (ONR), as well as the National Oceanic and Atmospheric Administration (NOAA). MOOS and IvP are currently funded by Code 31 at ONR, Dr. Don Wagner and Dr. Behzad Kamgar-Parsi. Testing and development of course work at MIT is further supported by Battelle, Dr. Robert Carnes. MOOS is additionally supported in the U.K. by EPSRC. Early development of IvP benefited from the support of the In-house Laboratory Independent Research (ILIR) program at the Naval Undersea Warfare Center in Newport RI.

1.4 The Software

The MOOS-IvP autonomy software is available at the following URL:

```
http://www.moos-ivp.org
```

Follow the links to *Software*. Instructions are provided for downloading the software from an SVN server with anonymous read-only access.

1.4.1 Building and Running the Software

This document is written to Release 13.2. After checking out the tree from the SVN server as prescribed at this link, the top level directory should have the following structure:

```
$ cd moos-ivp/
$ ls
MOOS@                   bin/                    include/
MOOS_V10                build/                  ivp/
README-LINUX.txt        build-ivp.sh*           lib/
README-OS-X.txt         build-moos.sh*          scripts/
README-WINDOWS.txt      configure-ivp.sh*
```

Note there is a `MOOS` directory and an `IvP` sub-directory. The `MOOS` directory is a symbolic link to a particular MOOS release checked out from the Oxford server. In the example above this is MOOS Version 10. The MOOS directory included with MOOS-IvP contains the MOOS Core middleware tree plus four other related trees distributed with MOOS:

- *core-moos*: Core middleware library, MOOSDB and MOOSApp superclass.
- *essential-moos*: Ubiquitious tools, pLogger, Antler, pShare and so on.
- *ui-moos*: GUI related tools, uMS, uPlayback and more.
- *geodesy-moos*: Tools for translating between local and earth coordinates.
- *matlab-moos*: Tools for interfacing with Matlab, including iMatlab.

The Core MOOS middlware library is very small, about 1.5Mb and has no external dependencies. The MOOS directory distributed with MOOS-IvP contains these five Oxford MOOS trees completely untouched other than a local re-naming of the folders and a build wrapper added to automate the build process. The use of a symbolic link is done to simplify the process of bringing in a new release from the Oxford server.

The build instructions are maintained in the README files and are probably more up to date than this document can hope to remain. In short building the software amounts to two steps - building MOOS and building IvP. Building MOOS is done by executing the build-moos.sh script:

```
$ cd moos-ivp
$ ./build-moos.sh
```

Alternatively one can go directly into the `MOOS` directory and configure options with `ccmake` and build with `cmake`. The script is included to facilitate configuration of options to suit local use. Likewise the IvP directory can be built by executing the `build-ivp.sh` script. The `MOOS` tree must

be built before building IvP. Once both trees have been built, the user's shell executable path must be augmented to include the two directories containing the new executables:

```
moos-ivp/MOOS/bin
moos-ivp/bin
```

At this point the software should be ready to run and a good way to confirm this is to run the example simulated mission in the missions directory:

```
$ cd moos-ivp/ivp/missions/alpha/
$ pAntler alpha.moos
```

Running the above should bring up a GUI with a simulated vehicle rendered. Clicking the DEPLOY button should start the vehicle on its mission. If this is not the case, some help and email contact links can be found at www.moos-ivp.org/support/, or emailing issues@moos-ivp.org.

1.4.2 Operating Systems Supported by MOOS and IvP

The MOOS software distributed by Oxford is well supported on Linux, Windows and Mac OS X. The software distributed by MIT includes additional MOOS utility applications and the IvP Helm and related behaviors. These modules are support on Linux and Mac OS X and the software compiles and runs on Windows but Windows support is limited.

1.4.3 Where to Get Further Information

Websites and Email Lists

There are two web sites - the MOOS web site maintained by Oxford University, and the MOOS-IvP web site maintained by MIT. At the time of this writing they are at the following URLs:

```
https://sites.google.com/site/moossoftware/

http://www.moos-ivp.org
```

What is the difference in content between the two web sites? As discussed previously, MOOS-IvP, as a set of software, refers to the software maintained and distributed from Oxford *plus* additional MOOS applications including the IvP Helm and library of behaviors. The software bundle released at moos-ivp.org does include the MOOS software from Oxford - usually a particular released version. For the absolute latest in the core MOOS software and documentation on Oxford MOOS modules, the Oxford web site is your source. For the latest on the core IvP Helm, behaviors, and MOOS tools distributed by MIT, the moos-ivp.org web site is the source.

There are two mailing lists open to the public. The first list is for MOOS users, and the second is for MOOS-IvP users. If the topic is related to one of the MOOS modules distributed from the Oxford web site, the proper email list is the "moosusers" mailing list. You can join the "moosusers" mailing list at the following URL:

```
https://lists.csail.mit.edu/mailman/listinfo/moosusers,
```

For topics related to the IvP Helm or modules distributed on the moos-ivp.org web site that are not part of the Oxford MOOS distribution (see the software page on moos-ivp.org for help in drawing the distinction), the "moosivp" mailing list is appropriate. You can join the "moosivp" mailing list at the following URL:

`https://lists.csail.mit.edu/mailman/listinfo/moosivp,`

Documentation

Documentation on MOOS can be found on the Oxford-maintained web site:

`https://sites.google.com/site/moossoftware/home/documentation`

This includes documentation on the MOOS architecture, programming new MOOS applications as well as documentation on several bread-and-butter applications such as `pAntler`, `pLogger`, `uMS`, `pShare`, `iRemote`, `iMatlab`, `pScheduler` and more. Documentation on the IvP Helm, behaviors and autonomy related MOOS applications not from Oxford can be found on the www.moos-ivp.org web site under the Documentation link. Below is a summary of documents:

2 Design Considerations of MOOS-IvP

The primary motivation in the design of MOOS-IvP is to build highly capable autonomous systems. Part of this picture includes doing so at a reduced short and long-term cost and a reduced time line. By "design" we mean both the choice in architectures and algorithms as well as the choice to make key modules for infrastructure, basic autonomy and advanced tools available to the public under an Open Source license. The MOOS-IvP software design is based on three architecture philosophies, (a) the backseat driver paradigm, (b) publish and subscribe autonomy middleware, and (c) behavior based autonomy. The common thread is the ability to separate the development of software for an overall system into distinct modules coordinated by infrastructure software made available to the public domain.

2.1 Public Infrastructure - Layered Capabilities

The central architecture idea of both MOOS and IvP is the separation of overall capability into separate and distinct modules. The unique contributions of MOOS and IvP are the methods used to *coordinate* those modules. A second central idea is the decision to make algorithms and software modules for infrastructure, basic autonomy and advanced tools available to the public under an Open Source license. The idea is pictured in Figure 1. There are three things in this picture - (a) modules that actually perform a function (the wedges), (b) modules that coordinate other modules (the center of the wheel), and (c) standard wrapper software use by each module to allow it to be coordinated (the spokes).

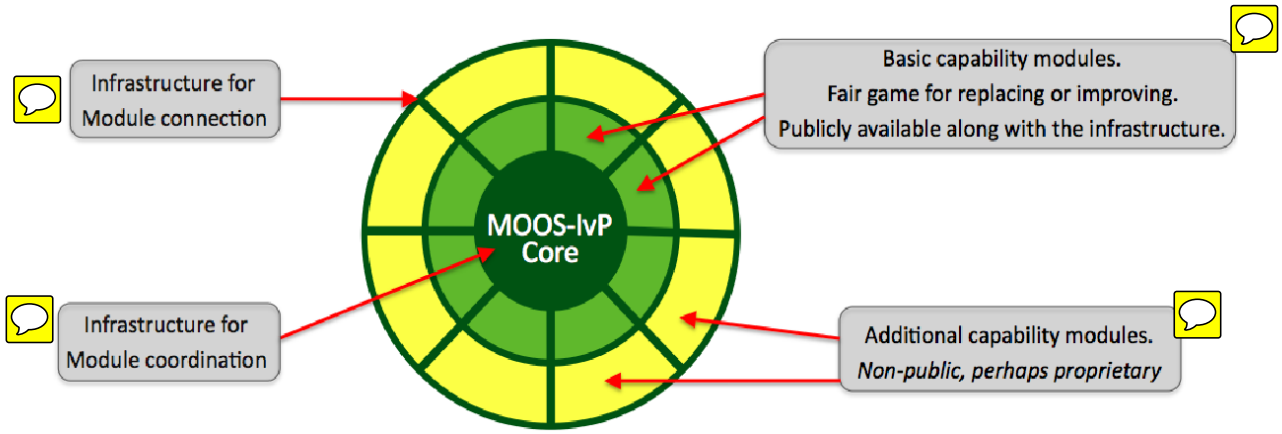


Figure 1: **Public Infrastructure - Layered Capabilities:** The center of the wheel represents MOOS-IvP Core. For MOOS this means the MOOSDB and the message passing and scheduling algorithms. For IvP this means the IvP helm behavior management and the multi-objective optimization solver. The wedges on the wheel represent individual modules - either MOOS processes or IvP behaviors. The spokes of the wheel represent the idea that each module inherits from a superclass to grab functionality key to plugging into the core. Each wedge or module contains a wrapper defined by the superclass that augments the function of the individual module. The darker wedges indicate publicly available modules and the lighter ones are modules added by users to augment the public set to comprise a particular fielded autonomy system.

The darker wedges in Figure 1 represent application modules (not infrastructure) that provide basic functionality and are publicly available. However, they do not hold any special immutable status. They can be replaced with a better version, or, since the source code is available, the

code of the existing module can be changed or augmented to provide a better or different version (hopefully with a different name - see the section on branching below). Later sections provide an overview of about 40 or so particular modules that are currently available. By modules we mean MOOS applications and IvP behaviors and the above comments hold in either case. The white wedges in Figure 1 represent the imaginable unimplemented modules or functionality. A particular fielded MOOS-IvP autonomy system typically is comprised of (a) the MOOS-IvP core modules, (b) *some* of the publicly available MOOS applications and IvP behaviors, and (c) additional perhaps non-public MOOS applications and IvP behaviors provided by one or more 3rd party developers.

The objective of the public-infrastructure/layered-capabilities idea is to strike an important balance - the balance between effective code re-use and the need for users to retain privacy regarding how they choose to augment the public codebase with modules of their own to realize a particular autonomy system. The benefits of code re-use are an important motivation in fundamental architecture decisions in both MOOS and IvP. The modules that comprise the public MOOS-IvP codebase described in this document represent over twenty work-years of development effort. Furthermore, certain core components of the codebase have had hundreds if not thousands of hours of usage on a dozen or so fielded platform types in a variety of situations. The issue of code re-use is discussed next.

2.2 Code Re-Use

Code re-use is critical, and starts with the ability to have a system comprised of separate but coordinated modules. They key technical hurdle is to achieve module separation without invoking a substantial hit on performance. In short, MOOS middleware is a way of coordinating separate processes running on a single computer or over several networked computers. IvP is a way of coordinating several autonomy behaviors running within a single MOOS process.

Factors Contributing to Code Re-use:

- *Freedom from proprietary issues.* Software serving as infrastructure shared by all components (MOOS processes and IvP behaviors) are available under an Open Source license. In addition many mature MOOS and IvP modules providing commonly needed capabilities are also publicly available. Proprietary or non-publicly released code may certainly co-exist with non-proprietary public code to comprise a larger autonomy system. Such a system would retain a strategic edge over competitors if desired, but have a subset of components common with other users.

- *Module independence.* Maintaining or augmenting a system comprised of a set of distinct modules can begin to break down if modules are not independent with simple easy-to-augment interfaces. Compile dependencies between modules need to be minimized or eliminated. The maintenance of core software libraries and application code should be decoupled completely from the issues of 3rd party additional code.

- *Simple well-documented interfaces.* The effort required to add modules to the code base should be minimized. Documentation is needed for both (a) using the publicly available applications and libraries, and (b) guiding users in adding their own modules.

- *Freedom to innovate.* The infrastructure does not put undue restrictions on how basic problems can be solved. The infrastructure remains agnostic to techniques and algorithms used

in the modules. No module is sacred and any module may be replaced.

Benefits of Code Re-Use:

- *Diversity of contributors.* Increasingly, an autonomy system contains many components that touch many areas of expertise. This would be true even for a vanilla use of a vehicle, but is compounded when considering the variety of sensors and missions and ways of exploiting sensors in achieving mission objectives. A system that allows for wide code re-use is also a system that allows module contributions from a wide set of developers or experts. This has a substantial impact on the issues mentioned below of lower cost, higher quality and reliability, and reduced development time line.

- *Lower cost.* One immediate benefit of code re-use is the avoidance of repeatedly re-inventing modules. A group can build capabilities incrementally and experts are free to concentrate on their area and develop only the modules that reflect their skill set and interests. Perhaps more important, code re-use gives the systems integrator *choices* in building a complete system from individual modules. Having choices leads to increased leverage in bargaining for favorable licensing terms or even non-proprietary terms for a new module. Favorable licensing terms arranged at the outset can lead to substantially lower long-term costs for future code maintenance or augmentation of software.

- *Higher performance capability.* Code re-use enhances performance capability in two ways. First, since experts are free to be experts without re-inventing the modules outside their expertise and provided by others, their own work is more likely to be more focused and efficient. They are likely to achieve a higher capability for a given a finite investment and given finite performance time. Second, since code re-use gives a systems integrator *choices*, this creates a meritocracy based on optimal performance-cost ratio of candidate software modules. The under-capable, more expensive module is less likely to diminish the overall autonomy capability if an alternative module is developed to offer a competitive choice. Survival of the fittest.

- *Higher performance reliability.* An important part of system reliability is testing. The more testing time and the greater diversity of testing scenarios the better. And of course the more time spent testing on physical vehicles versus simulation the better. By making core components of a codebase public and permitting re-use by a community of users, that community provides back an enormous service by simply using the software and complaining when or if something goes wrong. Certain core components of the MOOS-IvP codebase have had hundreds if not thousands of hours of usage on a dozen or so platform types in a variety of situations. And many more hours in simulation. Testing doesn't replace good coding practice or formal methods for testing and verifying correctness, but it complements those two aspects and is enhanced by code re-use.

- *Reduced development time line.* Code re-use means less code is being re-developed which leads to quicker overall system development. More subtly, since code re-use can provide a systems integrator choices and competition on individual modules, development time can be reduced as a consequent. An integrator may simply accept the module developed the quickest, or the competition itself may speed up development. If choices and competition result in

more favorable license agreements between the integrator and developer, this in itself may streamline agreements for code maintenance and augmentation in the long term. Finally, as discussed above, if code re-use leads to an element of community-driven bug testing, this will also quicken the pace in the evolution toward a mature and reliable autonomy system.

2.3 The Backseat Driver Design Philosophy

The key idea in the backseat driver paradigm is the separation between *vehicle control* and *vehicle autonomy*. The vehicle control system runs on a platform's main vehicle computer and the autonomy system runs on a separate payload computer. This separation is also referred to as the *mission controller - vehicle controller* interface. A primary benefit is the decoupling of the platform autonomy system from the actual vehicle hardware. The vehicle manufacturer provides a navigation and control system capable of streaming vehicle position and trajectory information to the main vehicle computer, and accepting a stream of autonomy decisions such as heading, speed and depth in return. Exactly how the vehicle navigates and implements control is largely unspecified to the autonomy system running in the payload. The relationship is depicted in Figure 2.

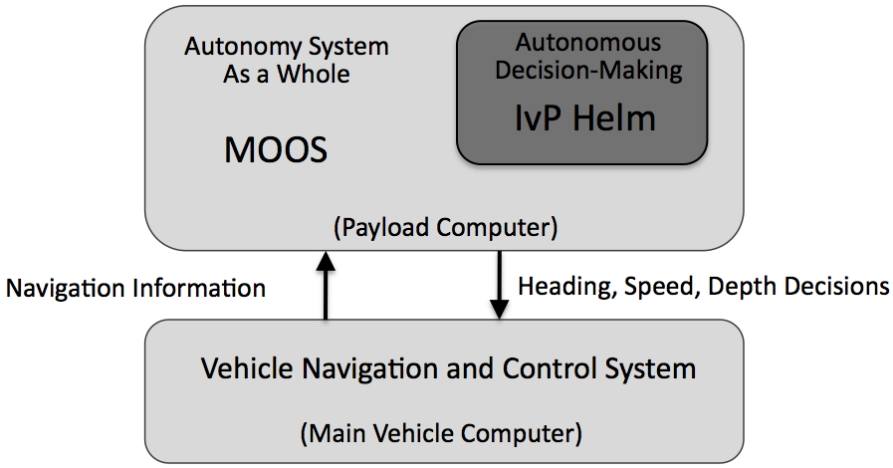


Figure 2: **The backseat driver paradigm**: The key idea is the separation of vehicle autonomy from vehicle control. The autonomy system provides heading, speed and depth commands to the vehicle control system. The vehicle control system executes the control and passes navigation information, e.g., position, heading and speed, to the autonomy system. The backseat paradigm is agnostic regarding how the autonomy system implemented, but in this figure the MOOS-IvP autonomy architecture is depicted.

The autonomy system on the payload computer consists of a set of distinct processes communicating through a publish-subscribe database called the MOOSDB (Mission Oriented Operating Suite - Database). One such process is an interface to the main vehicle computer, and another key process is the IvP Helm implementing the behavior-based autonomy system. The MOOS community is referred to as the "larger autonomy" system, or the "autonomy system as a whole" since MOOS itself is middleware, and actual autonomous decision making, sensor processing, contact management etc., are implemented as individual MOOS processes.

2.4 The Publish-Subscribe Middleware Design Philosophy and MOOS

MOOS provides a middleware capability based on the publish-subscribe architecture and protocol. Each process communicates with each other through a single database process in a star topology (Figure 3). The interface of a particular process is described by what messages it produces (publications) and what messages it consumes (subscriptions). Each message is a simple variable-value pair where the values are typically either string or numerical values such as (`STATE`, `"DEPLOY"`), or (`NAV_SPEED`, `2.2`). MOOS messages may also contain raw binary data for passing images for example.

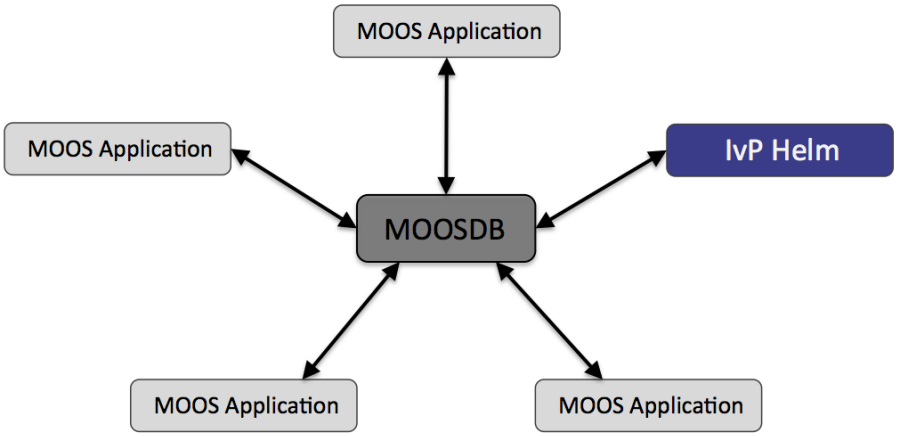


Figure 3: **A MOOS community**: is a collection of MOOS applications typically running on a single machine each with a separate process ID. Each process communicates through a single MOOS database process (the MOOSDB) in a publish-subscribe manner. Each process may be executing its inner-loop at a frequency independent from one another and set by the user. Processes may be all run on the same computer or distributed across a network.

The key idea with respect to facilitating code re-use is that applications are largely independent, defined only by their interface, and any application is easily replaceable with an improved version with a matching interface. Since MOOS Core and many common applications are publicly available along with source code under an Open Source GPL license, a user may develop an improved module by altering existing source code and introduce a new version under a different name. With MOOS-IvP 13.2 which includes MOOS V10, the MOOS libraries are distributed under an LGPL license, to allow the development and use of commercial MOOS applications alongside open source applications. The MOOSDB and MOOS applications remain under an GPL license. The term MOOS Core refers to (a) the MOOSDB application, and (b) the MOOS Application superclass that each individual MOOS application inherits from to allow connectivity to a running MOOSDB. Holding the MOOS Core part of the codebase constant between MOOS developers enables the plug-and-play nature of applications.

2.5 The Behavior-Based Control Design Philosophy and IvP Helm

The IvP Helm runs as a single MOOS application and uses a behavior-based architecture for implementing autonomy. Behaviors are distinct software modules that can be described as self-contained mini expert systems dedicated to a particular aspect of overall vehicle autonomy. The

helm implementation and each behavior implementation exposes an interface for configuration by the user for a particular set of missions. This configuration often contains particulars such as a certain set of waypoints, search area, vehicle speed, and so on. It also contains a specification of state spaces that determine which behaviors are active under what situations, and how states are transitioned. When multiple behaviors are active and competing for influence of the vehicle, the IvP solver is used to reconcile the behaviors (Figure 4).

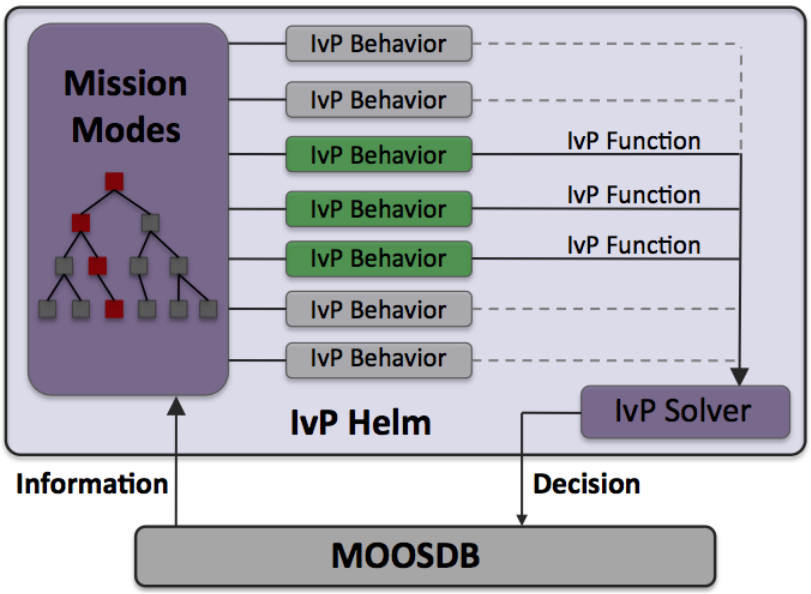


Figure 4: **The IvP Helm**: The helm is a single MOOS application running as the process pHelmIvP. It is a behavior-based architecture where the primary output of a behavior on each iteration is an IvP objective function. The IvP solver performs multi-objective optimization on the set of functions to find the single best vehicle action, which is then published to the MOOSDB. The functions are built and the set is solved on *each* iteration of the helm - typically one to four times per second. Only a subset of behaviors are active at any given time depending on the vehicle situation, and the state space configuration provided by the user.

The solver performs this coordination by soliciting an objective function, i.e., utility function, from each behavior defined over the vehicle decision space, e.g., possible settings for heading, speed and depth. In the IvP Helm, the objective functions are of a certain type - piecewise linearly defined - and are called IvP Functions. The solver algorithms exploit this construct to find a rapid solution to the optimization problem comprised of the weighted sum of contributing functions.

The concept of a behavior-based architecture is often attributed to [8]. Since then various solutions to the issue of action selection, i.e., the issue of coordinating competing behaviors, have been put forth and implemented in physical systems. The simplest approach is to prioritize behaviors in a way that the highest priority behavior locks out all others as in the Subsumption Architecture in [8]. Another approach is referred to as the potential fields, or vector summation approach (See [1], [11]) which considers the average action between multiple behaviors to be a reasonable compromise. These action-selection approaches have been used with reasonable effectiveness on a variety of platforms, including indoor robots, e.g., [1], [2], [15], [16], land vehicles, e.g., [17], and marine vehicles, e.g., [7], [9], [12], [18], [19]. However, action-selection via the identification of a single

highest priority behavior and via vector summation have well known shortcomings later described in [15], [16] and [17] in which the authors advocated for the use of multi-objective optimization as a more suitable, although more computationally expensive, method for action selection. The IvP model is a method for implementing multi-objective function based action-selection that is computationally viable in the IvP Helm implementation.

3 A Very Brief Overview of MOOS

MOOS is often described as autonomy *middleware* which implies that it is a kind of glue that connects a collection of applications where the real work happens. MOOS does indeed connect a collection of applications, of which the IvP Helm is one. MOOS is cross platform stand-alone and dependency free. It needs no other third-party libraries. Each application inherits a generic MOOS interface whose implementation provides a powerful, easy-to-use means of communicating with other applications and controlling the relative frequency at which the application executes its primary set of functions. Due to its combination of ease-of-use, general extendibility and reliability, it has been used in the classroom by students with no prior experience, as well as on many extended field exercises with substantial robotic resources at stake. To frame the later discussion of the IvP Helm, the basic issues regarding MOOS applications are introduced here. For further information on the original design of MOOS see [14].

3.1 Inter-process communication with Publish/Subscribe

MOOS has a star-like topology as depicted in Figure 3. Application within a MOOS community (a MOOSApp) have a connection to a single MOOS Database (called MOOSDB) lying at the heart of the software suite. All communication happens via this central server application. The network has the following properties:

- No peer-to-peer communication.
- Communication between the client and server is initiated by the client, i.e., the MOOSDB never makes a unsolicited attempt to contact a MOOSApp from out of the blue
- Each client has a unique name.
- A given client need have no knowledge of what other clients exist.
- One client does not transmit data to another - it can only be sent to the MOOSDB and from there to other clients. Modern versions of the library sport a sub-one millisecond latency when transporting multi-MB payloads between processes.
- The star network can be distributed over any number of machines running any combination of supported operating systems.
- The communications layer supports clock synchronization across all connected clients and in the same vein, can support "time acceleration" whereby all connected clients operate in an accelerated time stream - something that is very useful in simulations involving many processes distributed over many machines.
- data can be sent in small bites as "string" or "double" packets or in arbitrarily large binary packets.

3.2 Message Content

The communications API in MOOS allows data to be transmitted between the MOOSDB and a client. The meaning of that data is dependent on the role of the client. However the form of that data is not constrained by MOOS although for the sake on convenience MOOS does offer bespoke

Variable	Meaning
Name	The name of the data
String Value	Data in string format
Double Value	Numeric double float data
Source	Name of client that sent this data to the `MOOSDB`
Auxiliary	Supplemental message information, e.g., IvP behavior source
Time	Time at which the data was written
Data Type	Type of data (`STRING` or `DOUBLE` or `BINARY`)
Message Type	Type of Message (usually `NOTIFICATION`)
Source Community	The community to which the source process belongs

Table 1: The contents of MOOS message.

support for small "double" and string payloads. [1]. Data is packed into messages which contain other salient information shown in Table 1.

Often it is convenient to send data in string format for example the string `"Type=EST,Name=AUV,Pos=[3x1]3.4,6.` might describe the position estimate of a vehicle called "AUV" as a 3x1 column vector. This is human readable and does not require the sharing and synchronizing of header files to ensure both sender and recipient understand how to interpret data (as is the case with binary data). It is quite common for MOOS applications to communicate with string data in a concatenation of comma separated "name=value" pairs.

- Strings are human readable.
- All data becomes the same type.
- Logging files are human readable (they can be compressed for storage).
- Replaying a log file is simply a case of reading strings from a file and "throwing" them back at the MOOSDB in time order.
- The contents and internal order of strings transmitted by an application can be changed without the need to recompile consumers (subscribers to that data) - users simply would not understand new data fields but they would not crash.

The above are well understood benefits of sending self-explanatory ASCII data. However many applications use data types which do not lend themselves to verbose serialization to strings — think for example about camera image data being generated at 40Hz in full colour. At this point the need to send binary data is clear and of course MOOS supports it transparently (and the application pLogger supports logging and replaying it).

At this point it is up to the user to ensure that the binary data can be interpreted by all clients and that any and all perturbations to the data structures are distributed and compiled into each and every client. It is here that modern serialization tools such as "Google Protocol Buffers" find application. They offer a seamless way to serialize complex data structures into binary streams. Crucially they offer *forward* compatibility – it is possible to update and augment data structures with new fields in the comforting knowledge that all existing apps will still be able to interpret the data - they just won't parse the new additions.

[1] note that very early versions of MOOS only allowed data to be sent as strings or doubles - but this restriction is now long gone

3.3 Mail Handling - Publish/Subscribe - in MOOS

Each MOOS application is a client having a connection to the MOOSDB. This connection is made on the client side and the client manages a threaded machinery that coordinates the communication with the MOOSDB. This completely hides the intricacies and timings of the communications from the rest of the application and provides a small, well defined set of methods to handle data transfer. The application can:

1. Publish data - issue a notification on named data.
2. Register for notifications on named data.
3. Collect notifications on named data - reading mail.

3.3.1 Publishing Data

Data is published as a pair - a variable and value - that constitute the heart of a MOOS message described in Table 1. The client invokes the `Notify(VarName, VarValue)` command where appropriate in the client code. The above command is implemented both for string, double and binary values, and the rest of the fields described in Table 1 are filled in automatically. Each notification results in another entry in the client's "outbox", which in older versions of MOOS, is emptied the next time the `MOOSDB` accepts an incoming call from the client or in recent versions, is pushed instantaneously to all interested clients.

3.3.2 Registering for Notifications

Assume that a list of names of data published has been provided by the author of a particular MOOS application. For example, a application that interfaces to a GPS sensor may publish data called `GPS_X` and `GPS_Y`. A different application may register its interest in this data by subscribing or registering for it. An application can register for notifications using a single method `Register()` specifying both the name of the data and the maximum rate at which the client would like to be informed that the data has been changed. The latter parameter is specified in terms of the minimum possible time between notifications for a named variable. For example setting it to zero would result in the client receiving each and every change notification issued on that variable. MOOS V10 and later also supports "wildcard" subscriptions. For example a client can register for `"*:*"` to receive all messages from all other clients. Or `"GPS_*:?NAV"` to receive messages beginning with `"GPS_"` from any process with a four letter name ending in `"NAV"`.

3.3.3 Reading Mail

A client can enquire at any time whether it has received any new notifications from the `MOOSDB` by invoking the `Fetch` method. The function fills in a list of notification messages with the fields given in Table 1. Note that a single call to `Fetch` may result in being presented with several notifications corresponding to the same named data. This implies that several changes were made to the data since the last client-server conversation. However, the time difference between these similar messages will never be less than that specified in the `Register()` function described above. In typical applications the `Fetch` command is called on the client's behalf just prior to the `Iterate()` method, and the messages are handled in the user overloaded `OnNewMail()` method.

3.4 Overloaded Functions in MOOS Applications

MOOS provides a base class called `CMOOSApp` which simplifies the writing of a new MOOS application as a derived subclass. Beneath the hood of the `CMOOSApp` class is a loop which repetitively calls a function called `Iterate()` which by default does nothing. One of the jobs as a writer of a new MOOS-enabled application is to flesh this function out with the code that makes the application do what we want. Behind the scenes this overall loop in `CMOOSApp` is also checking to see if new data has been delivered to the application. If it has, another virtual function, `OnNewMail()`, is called. This is the function within which code is written to process the newly delivered data.

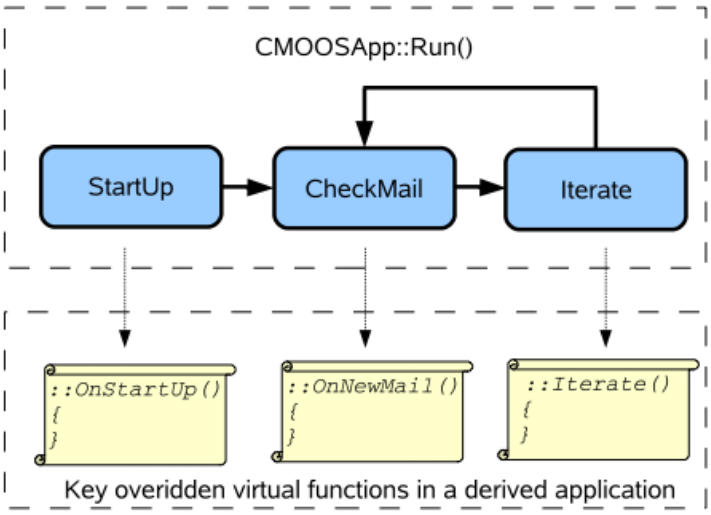


Figure 5: **Key virtual functions of the MOOS application base class**: The flow of execution once `Run()` has been called on a class derived from `CMOOSApp` . The scrolls indicate where users of the functionality of CMOOSApp will be writing new code that implements whatever it is that is wanted from the new applications. Note that it is not the case (as the above may suggest) that mail is polled for - in modern incantations of MOOS it is pushed to a client a synchronously `OnNewMail()` is called as soon as `Iterate()` is not running.

The roles of the three virtual functions in Figure 5 are discussed below. The `pHelmIvP` application does indeed inherit from `CMOOSApp` and overload these functions. The base class contains other virtual functions (`OnConnectToServer()` and `OnDisconnectFromServer()`) discussed in [14].

3.4.1 The Iterate() Method

By overriding the `CMOOSApp::Iterate()` function in a new derived class, the author creates a function from which the work that the application is tasked with doing can be orchestrated. In the `pHelmIvP` application, this method will consider the next best vehicle decision, typically in the form of deciding values for the vehicle heading, speed and depth. The rate at which `Iterate()` is called by the `SetAppFreq()` method or by specifying the `AppTick` parameter in a mission file. Note that the requested frequency specifies the maximum frequency at which `Iterate()` will be called - it does not guarantee that it will be called at the requested rate. For example if you write code in `Iterate()` that takes 1 second to complete there is no way that this method can be called at more than 1Hz. If you want to call `Iterate()` as fast as is possible simply request a frequency of zero - but you may want to reconsider why you need such a greedy application.

3.4.2 The OnNewMail() Method

Just before `Iterate()` is called, the `CMOOSApp` base class determines whether new mail is present, i.e., whether some other process has posted data for which the client has previously registered, as described above. If new mail is waiting, the `CMOOSApp` base class calls the `OnNewMail()` virtual function, typically overloaded by the application. The mail arrives in the form of a list of `CMOOSMsg` objects (see Table 1). The programmer is free to iterate over this collection examining who sent the data, what it pertains to, how old it is, whether or not it is string or numerical data and to act on or process the data accordingly. In recent versions of MOOS it is possible to have `OnNewMail()` called in a directly and rapidly in response to new mail being received by the back-end communications threads. This architecture allows for very rapid response times (sub ms) between a client posting data and it being received and handled by all interested parties.

3.4.3 The OnStartUp() Method

The `OnStartUp()` function is called just before the application enters into its own forever-loop depicted in Figure 5. This is the application that implements the application's initialization code, and in particular reads configuration parameters (including those that modify the default behavior of the CMOOSApp base class) from a file.

3.5 MOOS Mission Configuration Files

Every MOOS process can read configuration parameters from a mission file which by convention has a `.moos` extension. Traditionally MOOS processes share the same mission file to the maximum extent possible. For example, it is customary for there to be one common mission file for all MOOS processes running on a given machine. Every MOOS process has information contained in a configuration block within a `*.moos` file. The block begins with the statement

```
ProcessConfig = ProcessName
```

where `ProcessName` is the unique name the application will use when connecting to the MOOSDB. The configuration block is delimited by braces. Within the braces there is a collection of parameter statements, one per line. Each statement is written as:

```
ParameterName = value
```

where `value` can be any string or numeric value. All applications deriving from `CMOOSApp` inherit several important configuration options. The most important options for `CMOOSApp` derived applications are `CommsTick` and `AppTick`. The latter configures how often the communications thread talks to the `MOOSDB` and the former how often (approximately) `Iterate()` will be called.

Parameters may also be defined at the "global" level, i.e., not in any particular process' configuration block. Three parameters that are mandatory and typically found at the top of all `*.moos` files are: `ServerHost` naming the IP address associated with the MOOSDB server being launched with this file, `ServerPort` naming the port number over which the MOOSDB server is communicating with clients, and `Community` naming the community comprising the server and clients. An example is shown in lines 1-3 in Listing 4-A.

3.6 Launching Groups of MOOS Applications with Antler

Antler provides a simple and compact way to start a MOOS mission comprised of several MOOS processes, a.k.a., a MOOS *community*. For example if the desired mission file is `alpha.moos` then executing the following from a terminal shell:

```
$ cd moos-ivp/ivp/missions/s1_alpha
$ pAntler alpha.moos
```

will launch the required processes for the mission. It reads from its configuration block (which is declared as `ProcessConfig=ANTLER`) a list of process names that will constitute the MOOS community. Each process to be launched is specified with a line with the general syntax

```
Run = procname [ @ LaunchConfiguration ] [ MOOSName ]
```

where `LaunchConfiguration` is an optional comma-separated list of `parameter=value` pairs which collectively control how the process `procname` (for example `pHelmIvP`, or `pLogger` or `MOOSDB`) is launched. Exactly what parameters can be specified is outside the scope of this discussion. Antler looks through its entire configuration block and launches one process for every line which begins with the `RUN=` left-hand side. When all processes have been launched Antler waits for all of them to exit and then quits itself.

There are many more aspects of Antler not discussed here but can be found in the Antler documentation at the MOOS web site (see Section 1.4.3). These include hooks for altering the console appearance for each launched process, controlling the search path for specifying how executables are located on the host file system, passing parameters to launched processes, running multiple instances of a particular process, and using Antler to launch multiple distinct communities on a network.

3.7 Scoping and Poking the MOOSDB

An important tool for writing and debugging MOOS applications (and IvP Helm behaviors) is the ability for the user to interact with an active MOOS community and see the current values of particular MOOS variables (scoping the DB) and to alter one or more variables with a desired value (poking the DB). Below are listed tools for scoping and poking respectively. More information on each can be found on the Oxford or MIT web sites, or in in some instances, other parts of this document.

Tools for scoping the MOOSDB:

- uMS - A GUI-based tool written in FLTK and maintained and distributed from the Oxford website.
- uXMS - A terminal-based tool maintained and distributed from the MIT website
- uHelmScope - A terminal-based tool specialized for displaying information about a running instance of the helm, but it also contains a general-purpose scoping utility similar to uXMS. Distributed from the MIT website.

- MOOSDB http - MOOS allows the `MOOSDB` to be configured to run an http server on the current `MOOSDB` variable-value pairs, viewable through a web browser.


Tools for poking the MOOSDB:

- uMS - The GUI-based tool for scoping, listed above, also provides a means for poking. Distributed from the Oxford website.
- uPokeDB - A light-weight command-line tool for poking one or more variable-value pairs, with the option of scoping on the before and after values of the poked variable before exiting. Distributed from the MIT website.
- pMarineViewer - A GUI-based tool primarily used for rendering the paths of vehicles in 2D space on a Geo display, but also can be configured to poke the DB with variable-value pairs connected to buttons on the display. Distributed from the MIT website.
- *uTimerScript*: Allows the user to script a set of pre-configured pokes to a MOOSDB with each entry in the script happening after a speci- fied amount of time. Script may be paused or fast-forwarded. Events may also be configured with random values and happen randomly in a chosen window of time. Section 14.
- uTermCommand - A terminal-based tool for poking the DB with pre-defined variable-value pairs. The user can configure the tool to associate aliases (as short as a single character) to quickly poke the DB. Distributed from the MIT website.
- iRemote - A terminal-based tool for remote control of a robotic platform running MOOS. It can be configured to associate a pre-defined variable-value poke with any un-mapped key on the keyboard. Distributed from the Oxford website.

The above list is almost certainly not a complete list for scoping and poking a `MOOSDB`, but it's a decent start.

3.8 A Simple MOOS Application - pXRelay


The bundle of applications distributed from `www.moos-ivp.org` contains a very simple MOOS application called `pXRelay`. The `pXRelay` application registers for a single "input" MOOS variable and publishes a single "output" MOOS variable. It makes a single publication on the output variable for each mail message received on the input variable. The value published is simply a counter representing the number of times the variable has been published. By running two (differently named) versions of `pXRelay` with complementary input/output variables, the two processes will perpetuate some basic publish/subscribe handshaking. This application is distributed primarily as a simple example of a MOOS application that allows for some illustration of the following topics introduced up to this point:

- Finding and launching with `pAntler` example code distributed with the MOOS-IvP software bundle.

- An example mission configuration file.
- Scoping variables on a running MOOSDB with the `uXMS` tool.
- Poking the MOOSDB with variable-value pairs using the `uPokeDB` tool.
- Illustrating the `OnStartUp()`, `OnNewMail()`, and `Iterate()` overloaded functions of the `CMOOSApp` base class.

Besides touching on these topics, the collection of files in the `pXRelay` source code sub-directory is not a bad template from which to build your own modules.

3.8.1 Finding and Launching the pXRelay Example

The `pXRelay` example mission should be in the same directory tree containing the source code. See Section 1.4 on page 12. There is a single mission file, `xrelay.moos`:

```
moos-ivp/
   MOOS/
   ivp/
      missions/
         xrelay/
            xrelay.moos    <---- The MOOS file
```

To run this mission from a terminal window, simply change directories and launch:

```
$ cd moos-ivp/ivp/missions/xrelay
$ pAntler xrelay.moos
```

After `pAntler` has launched each process, there should be four open terminal windows, one for each `pXRelay` process, one for `uXMS`, and one for the MOOSDB itself.

3.8.2 Scoping the pXRelay Example with uXMS

Among the four windows launched in the example, the window to watch is the `uXMS` window, which should have output similar to the following (minus the line numbers):

Listing 1 - Example uXMS output after the pXRelay example is launched.

```
0  VarName            (S)ource       (T)ime     (C)ommunity  VarValue
1  ----------------   ----------     ---------  ----------   ----------- (73)
2  APPLES              n/a             n/a       n/a           n/a
3  PEARS               n/a             n/a       n/a           n/a
4  APPLES_ITER_HZ      pXRelay_APPLES 14.93     xrelay        24.93561
5  PEARS_ITER_HZ       pXRelay_PEARS  14.94     xrelay        24.93683
6  APPLES_POST_HZ      n/a             n/a       n/a           n/a
7  PEARS_POST_HZ       n/a             n/a       n/a           n/a
```

Initially the only thing that is changing in this window is the integer at the end of line 1 representing the number of updates written to the terminal. Here `uXMS` is configured to scope on the six variables shown in the `VarName` column. Column 2 shows which process last posted on the variable, column 3 shows when the last posting occurred, column 4 shows the community name from

which the post originated, and column 5 shows the current value of the variable. The "n/a" entries indicate that a process has yet to write to the given variable. For further info on the workings of uXMS see [5], or type 'h' to see the help menu.

There are two pXRelay processes running - one under the alias pXRelay_APPLES publishing the variable APPLES as its output variable, APPLES_ITER_HZ indicating the frequency in which the Iterate() function is executed, and APPLES_POST_HZ indicating the frequency at which the output variable is posted. There is likewise a pXRelay_PEARS process and the corresponding output variables.

3.8.3 Seeding the pXRelay Example with the uPokeDB Tool

Upon launching the pXRelay example, the only variables actively changing are the *_ITER_HZ variables (lines 4-5 in Listing 1) which confirm that the Iterate() loop in each process is indeed being executed. The output for the other variables in Listing 1 reflect the fact that the two processes have not yet begun handshaking. This can be kicked off by poking the APPLES (or PEARS) variable, which is the input variable for pXRelay_PEARS, by typing the following:

```
> cd moos-ivp/ivp/missions/xrelay
> uPokeDB xrelay.moos APPLES=1
```

The uPokeDB tool will publish to the MOOSDB the given variable-value pair APPLES=1. It also takes as an argument the mission file, xrelay.moos, to read information on where the MOOSDB is running in terms of machine name and port number. The output should look similar to the following:

Listing 2 - Example uPokeDB output after poking the MOOSDB with APPLES=1.

```
0  PRIOR to Poking the MOOSDB
1    VarName               (S)ource      (T)ime       VarValue
2    ----------------      ----------    ----------   -------------
3    APPLES
4
5
6  AFTER Poking the MOOSDB
7    VarName               (S)ource      (T)ime       VarValue
8    ----------------      ----------    ----------   -------------
9    APPLES                uPokeDB       40.19        1.00000"
```

The output of uPokeDB first shows the value of the variable prior to the poke, and then the value afterwards. Further information on the uPokeDB tool can be found in [5]. Once the MOOSDB has been poked as above, the pXRelay_PEARS application will receive this mail and, in return, will write to its output variable PEARS, which in turn will be read by pXRelay_APPLES and the two processes will continue thereafter to write and read their input and output variables. This progression can be observed in the uXMS terminal, which may look something like that shown in Listing 3:

Listing 3 - Example uXMS output after the pXRelay example is seeded.

```
0   VarName           (S)ource         (T)ime    (C)ommunity   VarValue
1   ----------------   ----------      --------  ----------    ----------- (221)
2   APPLES             pXRelay_APPLES  44.78     xrelay        151
3   PEARS              pXRelay_PEARS   44.74     xrelay        151
4   APPLES_ITER_HZ     pXRelay_APPLES  44.7      xrelay        24.90495
5   PEARS_ITER_HZ      pXRelay_PEARS   44.7      xrelay        24.90427
6   APPLES_POST_HZ     pXRelay_APPLES  44.79     xrelay        8.36411
7   PEARS_POST_HZ      pXRelay_PEARS   44.74     xrelay        8.36406
```

Upon each write to the MOOSDB the value of the variable is incremented by 1, and the integer progression can be monitored in the last column on lines 2-3. The APPLES_POST_HZ and PEARS_POST_HZ variables represent the frequency at which the process makes a post to the MOOSDB. This of course is different than (but bounded above by) the frequency of the Iterate() loop since a post is made within the Iterate() loop only if mail had been received prior to the outset of the loop. In a world with no latency, one might expect the "post" frequency to be exactly half of the "iterate" frequency. We would expect the frequency reported on lines 6-7 to be no greater than 12.5, and in this case values of about 8.4 are observed instead.

3.8.4 The pXRelay Example MOOS Configuration File

The mission file used for the pXRelay example, xrelay.moos is discussed here. This file is provided as part of the MOOS-IvP software bundle under the "missions" directory as discussed above in Section 3.8.1. It is discussed here in three parts in Listings 4-A through 4-C below.

The part of the xrelay.moos file provides three mandatory pieces of information needed by the MOOSDB process for launching. The MOOSDB is a server and on line 1 is the IP address for the machine, and line 2 indicates the port number where clients can expect to find the MOOSDB once it has been launched. Since each MOOSDB and the set of connected clients form a MOOS "community", the community name is provided on line 3. Note the xrelay community name in the xrelay.moos file and the community name in column 4 of the uXMS output in Listing 1 above.

Listing 4-A - The xrelay.moos *mission file for the pXRelay example.*

```
 1  ServerHost = localhost
 2  ServerPort = 9000
 3  Community  = xrelay
 4
 5  //------------------------------------------
 6  // Antler configuration  block
 7  ProcessConfig = ANTLER
 8  {
 9    MSBetweenLaunches = 200
10
11    Run = MOOSDB  @ NewConsole = true
12    Run = pXRelay  @ NewConsole = true ~ pXRelay_PEARS
13    Run = pXRelay  @ NewConsole = true ~ pXRelay_APPLES
14    Run = uXMS  @ NewConsole = true
15  }
```

The configuration block in lines 7-15 of xrelay.moos is read by the pAntler for launching the processes or clients of the MOOS community. Line 9 specifies how much time, in milliseconds, between the launching of processes. Lines 11-14 name the four MOOS applications launched in this example. On these lines, the component "NewConsole = true" determines whether a new console window will be opened for each process. Try changing them to false - only the uXMS window really needs to be open. The others merely provide a visual confirmation that a process has been launched. The "~ pXRelay_PEARS" component of lines 12 and 13 tell pAntler to launch these applications with the given alias. This is required here since each MOOS client needs to have a unique name, and in this example two instances of the pXRelay process are being launched.

In lines 17-39 in Listing 4-B below, the two pXRelay applications are configured. Note that the argument to ProcessConfig on lines 20 and 32 is the alias for pXRelay specified in the Antler configuration block on lines 12 and 13. Each pXRelay process is configured such that its incoming and

outgoing MOOS variables complement one another on lines 25-26 and 37-38. Note the AppTick parameter (see Section 3.4.1) is set to 25 in both configuration blocks, and compare with the observed frequency of the Iterate() function reported in the variables APPLES_ITER_HZ and PEARS_ITER_HZ in Listing 1. MOOS has done a pretty faithful job in this example of honoring the requested frequency of the Iterate() loop in each application.

Listing 4-B - The xrelay.moos mission file - configuring the pXRelay processes.

```
17  //--------------------------------------
18  // pXRelay config block
19
20  ProcessConfig = pXRelay_APPLES
21  {
22    AppTick       = 25
23    CommsTick     = 25
24
25    OUTGOING_VAR  = APPLES
26    INCOMING_VAR  = PEARS
27  }
28
29  //--------------------------------------
30  // pXRelay config block
31
32  ProcessConfig = pXRelay_PEARS
33  {
34    AppTick       = 25
35    CommsTick     = 25
36
37    INCOMING_VAR  = APPLES
38    OUTGOING_VAR  = PEARS
39  }
```

In the last portion of the xrelay.moos file, shown in Listing 4-C below, the uXMS process is configured. In this example, uXMS is configured to scope on the six variables specified on lines 54-59 to give the output shown in Listings 1 and 3. By setting the paused parameter on line 49 to false, the output of uXMS is continuously and automatically updated - in this case four times per second due to the rate of 4Hz specified in lines 46-47. The display_* parameters in lines 50-52 ensure that the output in columns 2-4 of the uXMS output is expanded. See [5] for further ways to configure the uXMS tool.

Listing 4-C - The xrelay.moos mission file for the pXRelay example - configuring uXMS*.*

```
41  //--------------------------------------
42  // uXMS config block
43
44  ProcessConfig = uXMS
45  {
46    AppTick    = 4
47    CommsTick  = 4
48
49    paused            = false
50    display_source    = true
51    display_time      = true
52    display_community = true
53
54    var  = APPLES
55    var  = PEARS
56    var  = APPLES_ITER_HZ
```

```
57    var  = PEARS_ITER_HZ
58    var  = APPLES_POST_HZ
59    var  = PEARS_POST_HZ
60  }
```

3.8.5 Suggestions for Further Things to Try with this Example

- Take a look at the `OnStartUp()` method in the `XRelay.cpp` class in the `pXRelay` module in the software bundle to see how the handling of parameters in the `xrelay.moos` configuration file are implemented, and the subscription for a MOOS variable.
- Take a look at the `OnNewMail()` method in the `XRelay.cpp` class in the `pXRelay` module in the software bundle to see how incoming mail is parsed and handled.
- Take a look at the `Iterate()` method in the `XRelay.cpp` class in the `pXRelay` module in the software bundle to see an example of a MOOS process that acts upon incoming mail and conditionally posts to the `MOOSDB`
- Try changing the `AppTick` parameter in *one* of the `pXRelay` configuration blocks in the `xrelay.moos` file, re-start, and note the resulting change in the iteration and post frequencies in the `uXMS` output.
- Try changing the `CommsTick` parameter in *one* of the `pXRelay` configuration blocks in the `xrelay.moos` file to something much lower than the `AppTick` parameter, re-start, and note the resulting change in the iteration and post frequencies in the `uXMS` output.

3.9 MOOS Applications Available to the Public

Below are very brief descriptions of MOOS applications in the public domain. This is by no means a complete list. It does not include applications outside MIT and Oxford, and it is not even a complete list of applications from those organizations. For a more in-depth tour of MOOS applications, see [5].

3.9.1 MOOS Modules from Oxford

- `pAntler`: A tool for launching a collection of MOOS processes given a mission file. See [14], [13]. Also, see Section 3.6.
- `pShare`: A tool that allows messages to pass between communities and allows for the renaming of messages as they are shuffled between communities. See [13].
- `pLogger`: A logger for recording the activities of a MOOS session. It can be configured to record a fraction of, or all publications of any number of MOOS variables. See [13].
- `pScheduler`: A simple tool for generating and responding to messages sent to the MOOSDB by processes in a MOOS community. See [13].
- `uMS`: A GUI-Based MOOS scope for monitoring one or more MOOSDBs. See [13].
- `uPlayback`: An FLTK-based, cross platform GUI application that can load in log files and replay them into a MOOS community as though the originators of the data were really running and issuing notifications. See [13].

- `iMatlab`: An application that allows Matlab to join a MOOS community - even if only for listening in and rendering sensor data. It allows connection to the MOOSDB and access to local serial ports. See [13].
- `iRemote`: A terminal-based tool for remote control of a robotic platform running MOOS. It can be configured to associate a pre-defined variable-value poke with any un-mapped key on the keyboard. See [13].
- `uMVS`: A multi-vehicle AUV simulator, capable of simulating any number of vehicles and acoustic ranging between them and acoustic transponders. The vehicle simulation incorporates a full 6 D.O.F vehicle model replete with vehicle dynamics, center of buoyancy / center of gravity geometry, and velocity dependent drag. The acoustic simulation is also fairly smart. It simulates acoustic packets propagating as spherical shells through the water column. See [13].

3.9.2 Mission Monitoring Modules

Mission monitoring modules aid the user in either keeping a high-level tab on the mission as it unfolds, or help the user analyze and debug a mission. In release 13.2 this includes two powerful new tools for appcast monitoring, `uMAC` and `uMACView`. The `pMarineViewer` has also been substantially augmented to support appcast viewing.

- *pMarineViewer*: GUI tool for rendering events in an area of vehicle operation. It repeatedly updates vehicle positions from incoming node reports, and will render several geometric types published from other MOOS apps. The viewer may also post messages to the MOOSDB based on user-configured keyboard or mouse events. Section 13 and [5].
- *uHelmScope*: A terminal-based (non-GUI) scope onto a running IvP Helm process, and key MOOS variables. It provides behavior summaries, activity states, and recent behavior postings to the MOOSDB. A very useful tool for debugging helm anomalies. Section 11 and [5].
- *uXMS*: A terminal-based (non GUI) tool for scoping a MOOSDB Users may precisely configure the set of variables they wish to scope on by naming them explicitly on the command line or in the MOOS configuration block. The variable set may also be configured by naming one or more MOOS proceses on which all variables published by those processes will be scoped. Users may also scope on the *history* of a single variable. See [5].
- *uProcessWatch*: This application monitors the presence of MOOS apps on a watch-list. If one or more are noted to be absent, it will be so noted on the MOOS variable `PROC_WATCH_SUMMARY`. uProcessWatch is appcast-enabled and will produce a succinct table summary of watched processes and the CPU load reported by the processes themselves. The items on the watch list may be named explicitly in the config file or inferred from the Antler block or from list of `DB_CLIENTS`. An application may be excluded from the watch list if desired. See [5].
- *uMAC*: The `uMAC` application is a utility for Monitoring AppCasts. It is launched and run in a terminal window and will parse appcasts generated within its own MOOS community or those from other MOOS communities bridged or shared to the local MOOSDB. The primary advantage of uMAC versus other appcast monitoring tools is that a user can remotely log into a vehicle via ssh and launch `uMAC` locally in a terminal. See [5].
- *uMACView*: A GUI tool for visually monitoring appcasts. It will parse appcasts generated within its own MOOS community or those from other MOOS communities bridged or shared to the local MOOSDB. Its capability is nearly identical to the appcast viewing capability

built into pMarineViewer. It was intended to be an appcast viewer for non-pMarineViewer users. See [5].

3.9.3 Mission Execution Modules

Mission execution modules participate directly in the proper execution of the mission rather than simply helping to monitor, plan or analyze the mission.

- *pNodeReporter*: A tool for collecting node information such as present vehicle position, trajectory and type, and posting it in a single report for sharing between vehicles or sending to a shoreside display. See [5].
- *pBasicContactMgr*: The contact manager deals with other known vehicles in its vicinity. It handles incoming reports perhaps received via a sensor application or over a communications link. Minimally it posts summary reports to the MOOSDB, but may also be configured to post alerts with user-configured content about one or more of the contacts. May be used in conjunction with the helm to spawn contact-related behaviors for collision avoidance, tracking, etc. Section 15 and [5].
- *pEchoVar*: A tool for subscribing for a variable and re-publishing it under a different name. It also may be used to pull out certain fields in string publications consisting of comma-separated parameter=value pairs, publishing the new string using different parameters. See [5].
- *pSearchGrid*: An application for storing a history of vehicle positions in a 2D grid defined over a region of operation. See [5].

3.9.4 Mission Simulation Modules

Mission simulation modules are used only in simulation. Many of the applications in the uField Toolbox may also be considered simulation modules, but they also have a use case involving simulated sensors on actual physical vehicles. The two modules below are purely for simulated vehicles.

- *uSimMarine*: A simple 3D vehicle simulator that updates vehicle state, position and trajectory, based on the present actuator values and prior vehicle state. Typical usage scenario has a single instance of `uSimMarine` associated with each simulated vehicle. See [5].
- *uSimCurrent*: A simple application for simulating the effects of water current. Based on local current information from a given file, it repeatedly reads the vehicle's present position and publishes a drift vector, presumably consumed by uSimMarine. See [5].

3.9.5 Modules for Poking the MOOSDB

Poking the MOOSDB is a common and often essential part of mission execution and/or command and control. The `pMarineViewer` tool also contains several methods for poking the MOOSDB on user command.

- *uPokeDB*: A command-line tool for poking a MOOSDB with variable-value pairs provided on the command line. It finds the MOOSDB via mission file provided on the command line, or the IP address and port number given on the command line. It will connect to the DB, show the value prior to poking, poke the DB, and wait for mail from the DB to confirm the result of the poke. See [5].

- *uTimerScript*: Allows the user to script a set of pre-configured pokes to a MOOSDB with each entry in the script happening after a speci- fied amount of time. Script may be paused or fast-forwarded. Events may also be configured with random values and happen randomly in a chosen window of time. See Section 14 and [5].
- *uTermCommand*: A terminal application for poking the MOOSDB with pre-defined variable-value pairs. A unique key may be associated with each poke. See [5].

3.9.6 The Alog Toolbox

The Alog Toolbox is set of offline tools for analyzing and manipulating alog files produces by the `pLogger` application distributed with the Oxford MOOS codebase.

- *alogscan*: A command line tool for reporting the contents of a given MOOS .alog file. See [5].
- *alogclip*: A command line tool that will create a new MOOS .alog file from a given .alog file by removing entries outside a given time window. See [5].
- *aloggrep*: A command line tool that will create a new MOOS .alog file by retaining only the given MOOS variables or sources from a given .alog file. See [5].
- *alogrm*: A command line tool that will create a new MOOS .alog file by removing the given MOOS variables or sources from a given .alog file. See [5].
- *alogview*: A GUI tool for analyzing a vehicle mission by plotting one or more vehicle trajectories on the operation area, while viewing a plot of any of the numerical values in the alog file(s). See [5].

3.9.7 The uField Toolbox

The uField Toolbox contains a number of tools for supporting multi-vehicle missions where each vehicle is connected to a shoreside community. This includes both simulation and real field experiments. It also contains a number of simulated sensors that run on offboard the vehicle on the shoreside.

- *pHostInfo*: Automatically detect the vehicle's host information including the IP addresses, port being used by the MOOSDB, the port being used by local pShare for UDP listening, and the community name for the local MOOSDB. Post these to facilitate automatic intervehicle communications in especially in multi-vehicle scenarios where the local IP address changes with DHCP. See [5].
- *uFldNodeBroker*: Typically run on a vehicle or simulated vehicle in a multi-vehicle context. Used for making a connection to a shoreside community by sending local information about the vehicle such as the IP address, community name, and port number being used by pShare for incoming UDP messages. Presumably the shoreside community uses this to know where to send outgoing UDP messages to the vehicle. See [5].
- *uFldShoreBroker*: Typically run in a shoreside community. Takes reports from remote vehicles describing how they may be reached. Posts registration requests to shoreside pShare to bridge user-provided list of variables out to vehicles. Upon learning of vehicle JAKE will create bridges `FOO_ALL` and `FOO_JAKE` to JAKE, for all such user-configured variables. See [5].

- *uFldNodeComms*: A shoreside tool for managing communications between vehicles. It has knowledge of all vehicle positions based on incoming node reports. Communications may be limited based on vehicle range, frequency of messages, or size of message. Messages may also be blocked based on a team affiliation. See [5].
- *uFldMessageHandler*: A tool for handling incoming messages from other nodes. The message is a string that contains the source and destination of the message as well as the MOOS variable and value. This app simply posts to the local MOOSDB the variable-value pair contents of the message. See [5].
- *uFldScope*: Typically run in a shoreside community. Takes information from user-configured set of incoming reports and parses out key information into a concise table format. Reports may be any report in the form of comma-separated parameter-value pairs. See [5].
- *uFldPathCheck*: Typically run in a shoreside community. Takes node reports from remote vehicles and calculates the current vehicle speed and total distance travelled and posts them in two concise reports. Odometry tallies may be re-set to zero by other apps. See [5].
- *uFldHazardSensor*: Typically run in a shoreside community. Configured with a set objects with a given x,y location and classification (hazard or beningn). The sensor simulator receives a series of requests from a remote vehicle. When sensor determines that an object is is within the sensor field of a requesting vehicle, it may or may not return a sensor detection report for the object, and perhaps also a proper classification. The odds of receiving a detection and proper classification depend on the sensor configuration and the user's preference for P_D/P_FA on the prevailing ROC curve. See [5].
- *uFldHazardMetric*: An application for grading incoming hazard reports, presumably generated by users of the uFldHazardSensor after exploring a simulated hazard field. See [5].
- *uFldHazardMgr*: The uFldHazardMgr is a strawman MOOS app for managing hazard sensor information and generation of a hazard report over the course of an autonomous search mission. See [5].
- *uFldBeaconRangeSensor*: Typically run in a shoreside community. Configured with one or more beacons with known beacon locations. Takes range requests from a remote vehicle and returns a range report indicating that vehicle's range to nearby beacons. Range requests may or may not be answered depending on range to beacon. Reports may have noise added and may or may not include beacon ID. See [5].
- *uFldContactRangeSensor*: Typically run in a shoreside community. Takes reports from remote vehicles, notes their position. Takes a range request from a remote vehicle and returns a range report indicating that vehicle's range to nearby vehicles. Range requests may or may not be answered dependent on inter-vehicle range. Reports may also have noise added to their range values. See [5].

4 A First Example with MOOS-IvP - the Alpha Mission

In this section a simple mission is described using the IvP Helm. This example is designed to run in simulation on a single desktop/laptop machine. The mission configuraiton files for this example are distributed with the source code. Information on how to find these files and launch this mission are described below in Section 4.1. In this example the vehicle simply traverses a set of pre-defined given waypoints and returns back to the launch position. The user may re-call the vehicle prematurely before completing the waypoints, and may subsequently command the vehicle to resume the waypoints at any time. By this example the objective is to touch the following issues:

- Launching a mission with a given mission (`.moos`) file and behavior (`.bhv`) file.
- Configuration of MOOS processes, including the IvP Helm, with a `.moos` file.
- Configuration of the IvP Helm (mission planning) with a `.bhv` file.
- Implementation of simple command and control with the IvP Helm.
- Interaction between MOOS processes and the helm during normal mission operation.

4.1 Where to Find, and How to Launch the Alpha Example Mission

The example mission should be in the same directory tree containing the source code (See Section 1.4). There are two files - a MOOS file, also mission file or `.moos` file, and a behavior file or `.bhv` file:

```
moos-ivp/
   MOOS/
   ivp/
      missions/
         s1_alpha/
            alpha.moos    <---- The MOOS file
            alpha.bhv     <---- The Behavior file
```

To run this mission from a terminal window, simply change directories and launch:

```
$ cd moos-ivp/ivp/missions/s1_alpha
$ pAntler alpha.moos
```

After `pAntler` has launched each process, the `pMarineViewer` window should be open and look similar to that shown in Figure 6. After clicking the `DEPLOY` button in the lower right corner the vehicle should start to traverse the shown set of waypoints.

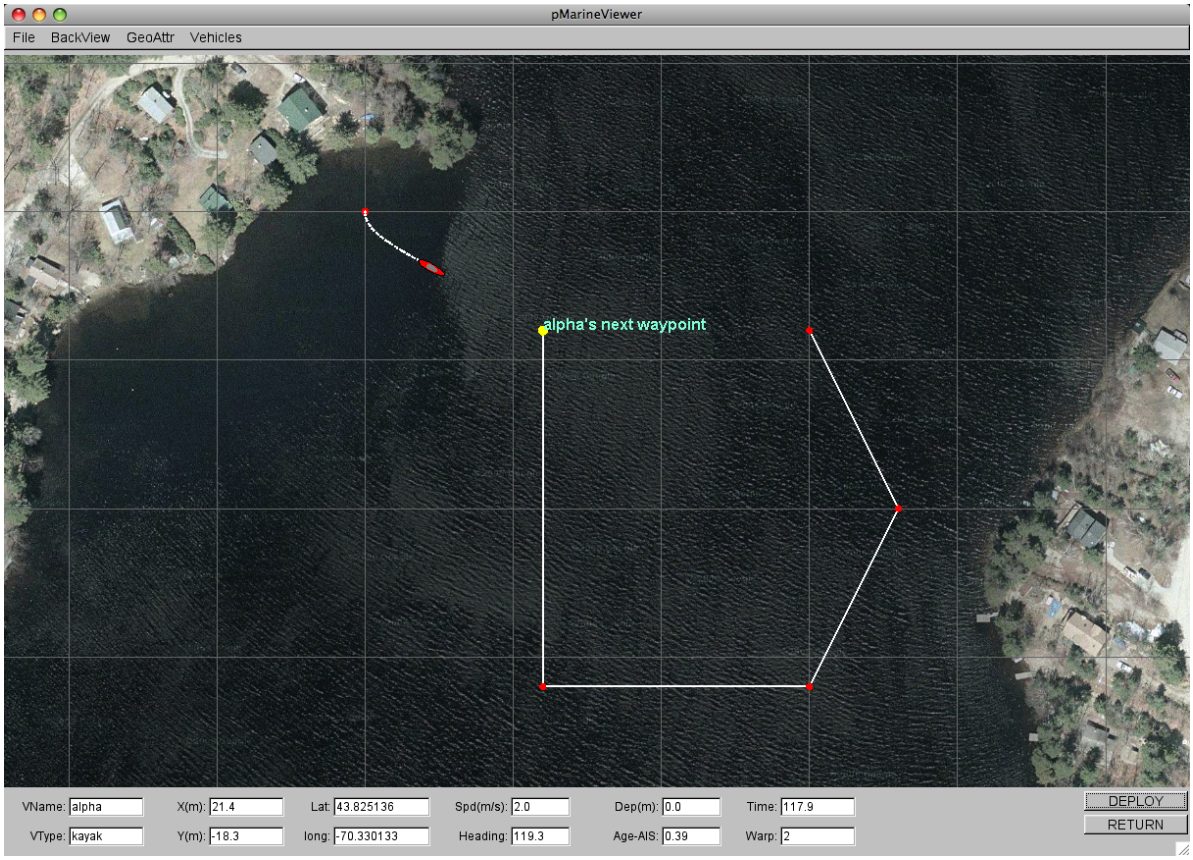


Figure 6: **The Alpha Example Mission - In the Surveying Mode**': A single vehicle is dispatched to traverse a set of waypoints and, upon completion, traverse to the waypoint (0,0) which is the launch point.

This mission will complete on its own with the vehicle returning to the launch point. Alternatively, by hitting the `RETURN` button at any time before the points have been traverse, the vehicle will change course immediately to return to the launch point, as shown in Figure 7. When the vehicle is returning as in the figure, it can be re-deployed by hitting the `DEPLOY` button again.

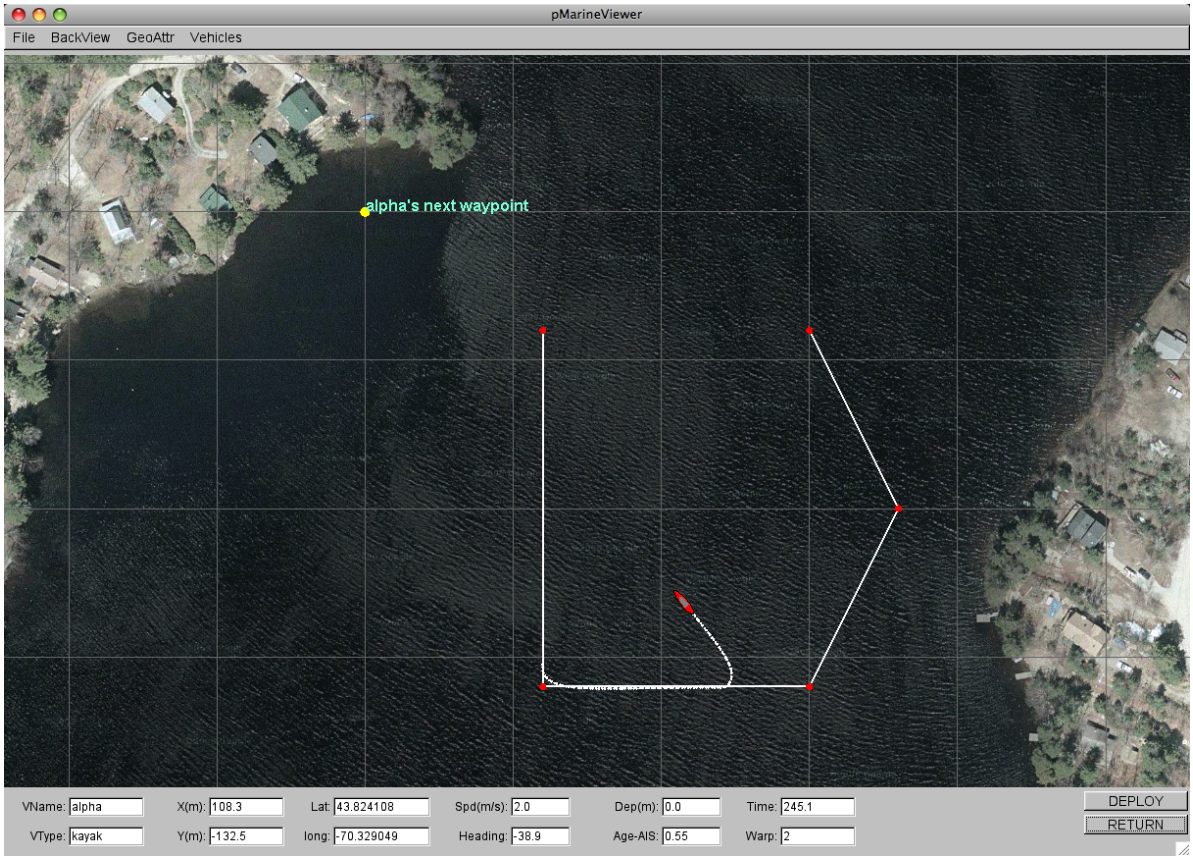


Figure 7: **The Alpha Example Mission - In the Returning Mode**': The vehicle can be commanded to return prior to the completion of its waypoints by the user clicking the `RETURN` button on the viewer.

The vehicle in this example is configured with two basic waypoint behaviors. Their configuration with respect to the points traversed and when each behavior is actively influencing the vehicle, is discussed next.

4.2 A Closer Look at the Behavior File used in the Alpha Example Mission

The mission configuration of the helm behaviors is provided in a *behavior file*, and the complete behavior file for the example mission is shown in Listing 5. Behaviors are configured in blocks of parameter-value pairs - for example lines 6-17 configure the waypoint behavior with the five waypoints shown in the previous two figures. This is discussed in more detail in Section 6.3.

Listing 5: The behavior file for the Alpha example.

```
0  initialize   DEPLOY = false
1  initialize   RETURN = false
2
3  //------------------------------------------
4  Behavior = BHV_Waypoint
5  {
6    name      = waypt_survey
7    pwt       = 100
```

```
 8    condition = RETURN = false
 9    condition = DEPLOY = true
10    endflag   = RETURN = true
11    perpetual = true
12
13            lead = 8
14     lead_damper = 1
15           speed = 2.0   // meters per second
16          radius = 4.0
17       nm_radius = 10.0
18          points = 60,-40:60,-160:150,-160:180,-100:150,-40
19          repeat = 1
20  }
21
22  //------------------------------------------------
23  Behavior = BHV_Waypoint
24  {
25    name       = waypt_return
26    pwt        = 100
27    condition  = RETURN = true
28    condition  = DEPLOY = true
29    perpetual  = true
30    endflag    = RETURN = false
31    endflag    = DEPLOY = false
32
33         speed = 2.0
34        radius = 2.0
35     nm_radius = 8.0
36         point = 0,0
37  }
```

The parameters for each behavior are separated into two groups. Parameters such as `name`, `priority`, `condition` and `endflag` are parameters defined generally for all IvP behaviors. Parameters such as `speed`, `radius`, and `points` are defined specifically for the Waypoint behavior. A convention used in `.bhv` files is to group the general behavior parameters separately at the top of the configuration block.

In this mission, the vehicle follows two sets of waypoints in succession by configuring two instances of a basic waypoint behavior. The second waypoint behavior (lines 23-37) contains only a single waypoint representing the vehicle launch point (0,0). It's often convenient to have the vehicle return home when the mission is completed - in this case when the first waypoint behavior has reached its last waypoint. Although it's possible to simply add (0,0) as the last waypoint of the first waypoint behavior, it is useful to keep it separate to facilitate recalling the vehicle pre-maturely at any point after deployment.

Behavior conditions (lines 8-9, 27-28), and endflags (line 10, lines 30-31) are primary tools for coordinating separate behaviors into a particular mission. Behaviors will not participate unless each of its conditions are met. The condtions are based on current values of the MOOS variables involved in the condition. For example, both behaviors will remain idle unless the variable `DEPLOY` is set to `true`. This variable is set initially to be `false` by the initialization on line 0, and is toggled by the `DEPLOY` button on the `pMarineViewer` GUI shown in Figures 6 and 7. The `pMarineViewer` MOOS application is one option for a command and control interface to the helm. The MOOS variables in the behavior conditions in Listing 5 do not care which process was responsible for setting the value. Endflags are used by behaviors to post a MOOS variable and value when a behavior has reached a completion. The notion of completion is different for each behavior and some behaviors have no notion of completion, but in the case of the waypoint behavior, completion is declared when the

last waypoint is reached. In this way, behaviors can be configured to run in a sequence, as in this example, where the returning waypoint behavior will have a necessary condition (line 27) met when the surveying behavior posts its endflag on line 10.

4.3 A Closer Look at the MOOS Applications in the Alpha Example Mission

Running the example mission involves five other MOOS applications in addition to the IvP helm. In this section we take a closer look at what those applications do and how they are configured. The full MOOS file, `alpha.moos`, used to run this mission is given in full in the appendix. An overview of the situation is shown in Figure 8.

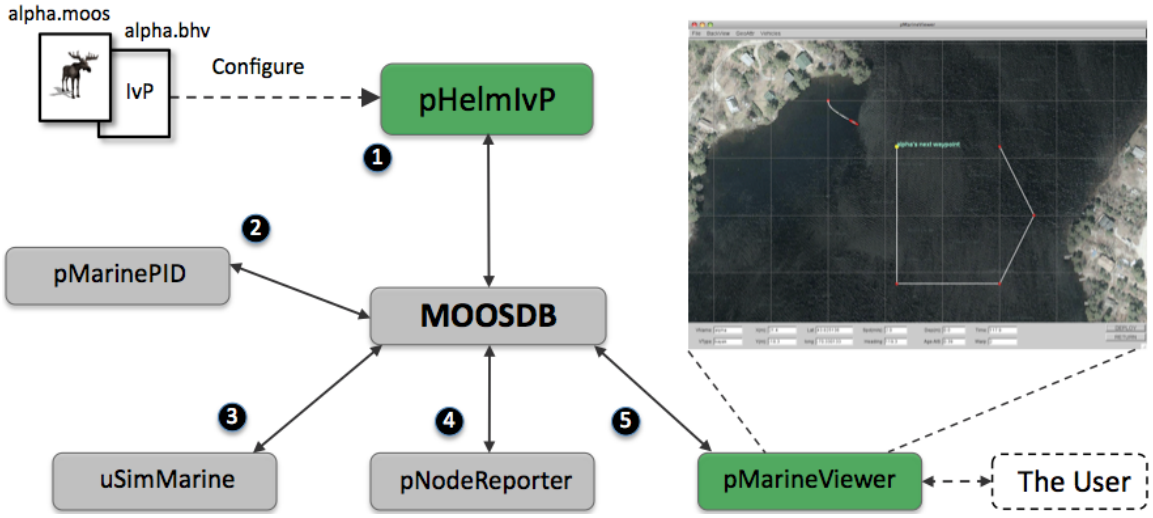


Figure 8: **The MOOS processes in the example "alpha" mission**: In (1) The helm produces a desired heading and speed. In (2) the PID controller subscribes for the desired heading and speed and publishes actuation values. In (3) the simulator grabs the actuator values and the current vehicle pose and publishes a set of MOOS variables representing the new vehicle pose. In (4) all navigation output is wrapped into a single node-report string to be consumed by the helm and the GUI viewer. In (5) the pMarineViewer grabs the node-report and renders a new vehicle posision. The user can interact with the viewer to write limited commmand and control variables to the MOOSDB.

4.3.1 Antler and the Antler Configuration Block

The `pAntler` tool is used to orchestrate the launching of all the MOOS processes participating in this example. From the command line, `pAntler` is run with a single argument the `.moos` file. As it launches processes, it hands each procoess a pointer to this same MOOS file. The Antler configuration block in this example looks like:

Listing 6 - An example Antler configuration block for the Alpha mission.

```
0   ProcessConfig = ANTLER
1  {
2    MSBetweenLaunches = 200
3
4    Run = MOOSDB          @ NewConsole = false
```

```
 5    Run = uSimMarine      @ NewConsole = false
 6    Run = pNodeReporter   @ NewConsole = false
 7    Run = pMarinePID      @ NewConsole = false
 8    Run = pMarineViewer   @ NewConsole = false
 9    Run = pHelmIvP        @ NewConsole = false
10  }
```

The first parameter on line 2 specifies how much time should be left beteen the launching of each process. Lines 4-9 specify which processes to launch. The MOOSDB is typically launched first. The `NewConsole` switch on each line determines whether a new console window should be opened with each process. Try switching one or more of these to `true` as an experiment.

4.3.2 The pMarinePID Application

The `pMarinePID` application implements a simple PID controller which produces values suitable for actuator control based on inputs from the helm. In simulation the output is consumed by the vehicle simulator rather than the vehicle actuatiors.

In short: The `pMarinePID` application typically gets its info from `pHelmIvP`; produces info consumed by `uSimMarine` or actuator MOOS processes when not running in simulation.

Subcribes to: `DESIRED_HEADING`, `DESIRED_SPEED`.

Publishes to: `DESIRED_RUDDER`, `DESIRED_THRUST`.

4.3.3 The uSimMarine Application and Configuration Block

The `uSimMarine` application is a very simple vehicle simulator that considers the current vehicle pose and actuator commands and produces a new vehicle pose. It can be initialized with a given pose as shown in the configuration block used in this example, shown in Listing 7:

Listing 7 - An example uSimMarine configuration block for the Alpha mission.

```
 0  ProcessConfig = uSimMarine
 1  {
 2    AppTick  = 10
 3    CommsTick = 10
 4
 5    START_X       = 0
 6    START_Y       = 0
 7    START_SPEED   = 0
 8    START_HEADING = 180
 9    PREFIX        = NAV
10  }
```

In short: The `uSimMarine` application typically gets its info from `pMarinePID`; produces info consumed by `pNodeReporter` and itself on the next iteration of `uSimMarine`.

Subcribes to: `DESIRED_RUDDER`, `DESIRED_THRUST`, `NAV_X`, `NAV_Y`, `NAV_SPEED`, `NAV_HEADING`.

Publishes to: `NAV_X`, `NAV_Y`, `NAV_HEADING`, `NAV_SPEED`.

4.3.4 The pNodeReporter Application and Configuration Block

An Automated Information System (AIS) is commonplace on many larger marine vessels and is comprised of a transponder and receiver that broadcasts one's own vehicle ID and pose to other nearby vessels equiped with an AIS receiver. It periodically collects all latest pose elements, e.g., latitutude and longitude position and latest measured heading and speed, and wraps it up into a single update to be broadcast. This MOOS process collects pose information by subscribing to the `MOOSDB` for `NAV_X`, `NAV_Y`, `NAV_HEADING`, `NAV_SPEED`, and `NAV_DEPTH` and wraps it up into a single MOOS variable called `NODE_REPORT_LOCAL`. This variable in turn can be subscribed to another MOOS process connected to an actual serial device acting as an AIS transponder. For our purposes, this variable is also subscribed to by pMarineViewer for rendering a vehicle pose sequence.

In short: The `pNodeReporter` application typically gets its info from `uSimMarine` or otherwise on-board navigation systems such as GPS or compass; produces info consumed by `pMarineViewer` and instances of `pHelmIvP` running in other vehicles or simulated vehicles.

Subcribes to: `NAV_X`, `NAV_Y`, `NAV_SPEED`, `NAV_HEADING`.

Publishes to: `NODE_REPORT_LOCAL`

4.3.5 The pMarineViewer Application and Configuration Block

The `pMarineViewer` is a MOOS process that subscribes to the MOOS variable `NODE_REPORT_LOCAL` and `NODE_REPORT` which contains a vehicle ID, pose and timestamp. It renders the updated vehicle(s) position. It is a multi-threaded process to allow both communication with MOOS and let the user pan and zoom and otherwise interact with the GUI. It is non-essential for vehicle operation, but essential for visually confirming that all is going as planned.

In short: The `pMarineViewer` application typically gets its info from `pNodeReporter` and `pHelmIvP`; produces info consumed by `pHelmIvP` when configured to have command and control hooks (as in this example).

Subcribes to: `NODE_REPORT`, `NODE_REPORT_LOCAL`, `VIEW_POINT`, `VIEW_SEGLIST`, `VIEW_POLYGON`, `VIEW_MARKER`.

Publishes to: Depends on configuration, but in this example: `DEPLOY`, `RETURN`.

5 The IvP Helm as a MOOS Application

In this section the helm is discussed in terms of its identity as a MOOS application - its MOOS configuration parameters, its `Iterate()` loop, its output to the console, and its output in terms of publications to other applications running in the MOOS community. The *helm state* and *all-stop status* are also introduced since they are the highest level descriptions regarding helm activity.

5.1 Overview

The IvP Helm is implemented as the MOOS module called `pHelmIvP`. On the surface it is similar to any other MOOS application - it runs as a single process that connects to a running MOOSDB process interfacing solely by a publish-subscribe interface, as depicted in Figure 9. It is configured from a behavior file, or `.bhv` file, in addition to the MOOS file used to configure other MOOS applications. The helm primarily publishes a steady stream of information that drives the platform, typically regarding the desired heading, speed or depth. It may also publish information conveying aspects of the autonomy state that may be useful for monitoring, debugging or triggering other algorithms either within the helm or in other MOOS processes. The helm can be configured to generate decisions over virtually any user-defined decision space.

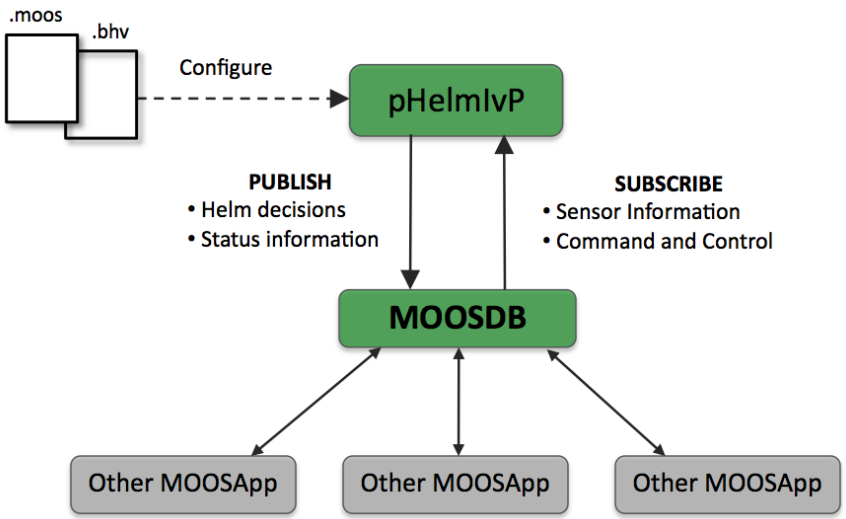


Figure 9: **The `pHelmIvP` MOOS application**: The IvP Helm is implemented as the MOOS application `pHelmIvP`. The helm is configured with two files - the mission file and behavior file. Once launched it connects to the MOOSDB along with other MOOS applications performing other functions. Information flowing into the helm include both sensor information and command and control inputs. The helm produces commands for maneuvering the vehicle along with other status information produced by active behaviors.

The helm subscribes for sensor information or any other information it needs to make decisions. This information includes navigation information regarding the platform's current position and trajectory, information regarding the position or state of other vehicles, or environmental information. The information it subscribes for is prescribed by the behaviors themselves, configured

in the `.bhv` file. In addition to sensor information, the helm also receives some level of command and control information. For example, in some marine vehicle configurations, one of the "Other MOOSApp" modules in the figure is a driver for an acoustic modem over which command and control information may be relayed.

The helm has a couple informative high-level state descriptions, the *helm state* and the *all-stop status*, that may be compared to the operation of an automobile. Launching the `pHelmIvP` MOOS process is analagous to turning on the car's engine. Putting the helm in the `DRIVE` mode is like shifting the car from "Park" to "Drive". And the all-stop status refers whether or not the car is breaking to a stop. The analogy is summarized below.

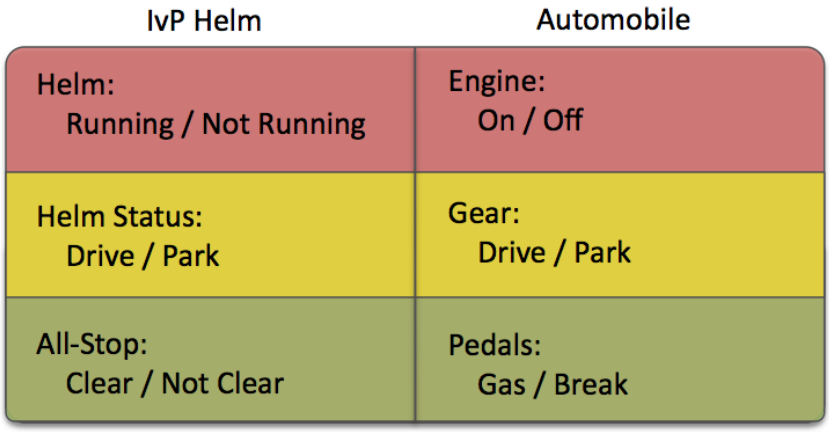


Figure 10: **The Helm/Automobile Analogy**: The helm and its high-level state descriptions, the *helm state* and *all-stop status*, have analogies with the operation of an automobile.

5.2 The Helm State

The highest level interface with the helm is reflected in the *helm state*. The helm state may have one of two values, *park* or *drive* (except when using a *standby helm* described in Section 5.7) In the *drive* mode, the helm is likely in the process of executing a mission. In the *park* mode, the helm is waiting, likely because it is being asked to wait, but also because the helm may have noticed something wrong (generated an all-stop) and subsequently put itself into *park* on its own, awaiting a the chance to return to the *drive* state. In this section we discuss (a) how the helm state is changed, (b) what it is going on in the helm when it is parked, and (c) how the helm state is initialized at start-up. At any point in time after the helm is launched, the helm will post the MOOS variable `IVPHELM_STATE` with either the value `"DRIVE"` or `"PARK"`. This is posted on each iteration and registering for this mail is the manner recommended by which other MOOS applications monitor the helm's heart beat.

Helm State Transitions

The helm state may be transitioned by writing to the MOOS variable `MOOS_MANUAL_OVERIDE`. As Figure 11 depicts, a value of `false`, which is case insensitive, transitions the helm state into `DRIVE`. A value of `true` puts it into the `PARK`. When the helm transitions from `DRIVE` to `PARK` it makes *one* more publication to the helm decision variables, each with a value zero. This is referred to as the

production of an *all-stop posting*, discussed in more detail in a later section.

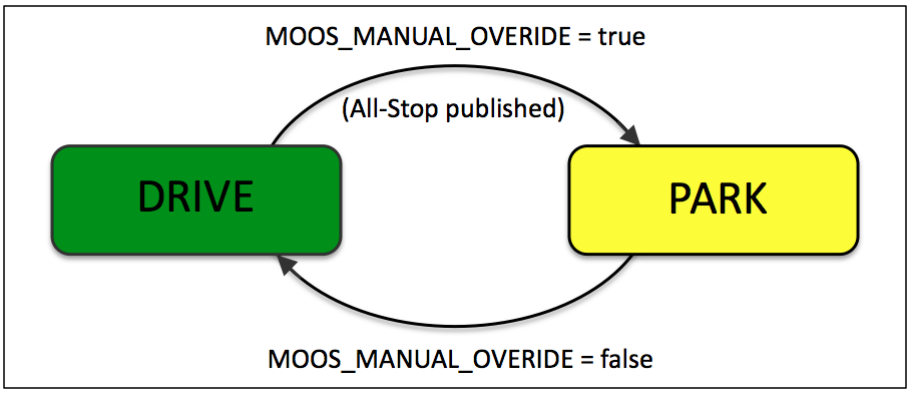


Figure 11: **The Helm State of the IvP Helm**: The helm state has a value of either `PARK` or `DRIVE`, depending on both how the helm is initialized and the mail received by the helm after start-up on the variable `MOOS_MANUAL_OVERIDE`. The helm may also park itself if an all-stop event has been detected.

The variable `MOOS_MANUAL_OVERIDE` contains the mis-spelling of "override". However, it is a variable that has some legacy presence in other MOOS applications such as `iRemote`. To avoid a situation where there is an attempt to override the helm, but the request is ignored because of a (proper) spelling, the helm will also respect transition requests on the properly spelled variable `MOOS_MANUAL_OVERRIDE`. This has the drawback however that these two variables could conceivably have different values in the MOOSDB. This is not a problem but could be confusing for someone trying to infer the helm state by opening a scope on the MOOSDB, on either the wrong variable or the two disagreeing variables. In this case the helm state would be aligned with the variable with the most recent publication time stamp. In any event, the best way to monitor the helm state is to scope on the MOOS variable `IVPHELM_STATE`, published by the helm itself, or use the `uHelmScope` tool.

The helm may also automatically transition itself from the `DRIVE` to the `PARK` state (but never the other way around), by posting and all-stop event. All-stop events and the helm are discussed separately in Section 5.3. All-stop events are generated by the helm upon finding that one or more possible error conditions have been detected during the normal execution of the helm iteration. If the helm parks due to an all-stop, it may be returned to the `DRIVE` state by another MOOS client posting `MOOS_MANUAL_OVERIDE=false`, but this is no guarantee that the helm wouldn't just park again immediately if the same condition persists that caused the all-stop event.

What Is and Isn't Happening when the Helm is Parked

When the helm state is `PARK`, the MOOS application loop depicted in Figure 5 on page 25 carries on. The `OnNewMail()` continues to be called and new mail is read and dealt with exactly as it would if the helm state were `DRIVE`. The `Iterate()` loop, however, is truncated to virtually a no-op, with the only action being the output of a heartbeat character to the console if the helm is configured to do so (Section 5.5). No behavior code is called whatsoever. The helm iteration counter, a key

index in the uHelmScope output, is also suspended despite the fact that technically the Iterate() loop continues to be called.

Initializing the Helm State at Process Launch Time

The helm, by default, is configured to be initiallly in PARK upon start-up. By setting the parameter start_in_drive=true in the mission file configuration block, the helm will indeed be in the DRIVE upon start-up. This feature was found to have practical use in UUV operations to allow for rebooting of the autonomy computer to automatically launch the helm, in DRIVE and ready to accept field commands. This feature should be used with caution, and it may be phased out in a later software release.

5.3 Helm All-Stop Events and All-Stop Status

An *all-stop event* is something that brings the vehicle to a full-stop, with typically zero-speed, zero-depth commanded. The *all-stop status* is simply a string describing *why* the vehicle is at an all-stop. Sometimes an all-stop indicates a problem, e.g., missing critical sensor information. Sometimes a vehicle may just stop as part of the mission, e.g., coming to the surface for a GPS fix. The all-stop status message can be used to discern the two types of situations. In the automobile analogy, an all-stop is equivalent to hitting the car's breaks with the intent to stop completely. An all-stop event will result in the following:

- Zero-values will be posted for all decision variables, DESIRED_SPEED=0, DESIRED_DEPTH=0, etc.
- The helm will possibly transition into PARK. By default the helm is configured to remain in the DRIVE upon an all-stop event, but if configured instead with park_on_allstop = true, the helm will indeed park upon an all-stop.
- The reason for the all-stop will be posted to MOOS variable IVPHELM_ALLSTOP. The value of this variable will be "clear" if there are no all-stop events that have occurred since the helm has entered the DRIVE state.

The reasons for all-stop may be:

- No behaviors are active. The helm has absolutely no opinion about any of its decision variables. In this case, the following would be posted: IVPHELM_ALLSTOP ="NoIvPFunctions".
- Some behaviors are active, but decisions are missing on one or more mandatory decision variables. In this case, the following would be posted: IVPHELM_ALLSTOP ="MissingDecVars".
- When the vehicle is parked due to manual override, the following would be posted: IVPHELM_ALLSTOP ="ManualOverride".
- One of the behaviors has determined an all-stop is warranted for some reason. For example, a waypoint behavior that cannot determine own-platform's current position would declare an all-stop. In this case, the following would be posted: IVPHELM_ALLSTOP ="BehaviorError".

To gain further insight on an all-stop caused by a behavior error, the nature of the error is expressed in a separate posting to the MOOS BHV_ERROR variable. It's possible that more than one behavior error occurred in the helm iteration where the all-stop event occurred, in which case there would be multiple postings to the BHV_ERROR variable. When the vehicle is in DRIVE and operating free of an all-stop, the all-stop status is reflected by IVPHELM_ALLSTOP ="clear".

Experiment 1: Noting the helm state and all-stop status in the Alpha mission.

Issues Explored: (1) The relationship between the MOOS_MANUAL_OVERIDE variable and the helm status and all-stop status. (2) How to visually confirm this relationship.

- Try running the Alpha mission again from Section 4. Note that when the simulation is first launched, before the DEPLOY button is hit in the pMarineViewer window, the label next to the vehicle says "alpha (PARK) (ManualOverride)"
- Try opening a uXMS scope to scope on a few key variables:

  ```
  $ uXMS alpha.moos MOOS_MANUAL_OVERIDE IVPHELM_STATE IVPHELM_ALLSTOP
  ```

  Note that MOOS_MANUAL_OVERIDE = "true", IVPHELM_STATE = "PARK", and IVPHELM_ALLSTOP = "ManualOverride".
- Deploy the vehicle by hitting the DEPLOY button in the pMarineViewer window. Note the changes in both the vehicle label in the GUI, and the values of the variables in the uXMS scope. Note that when the all-stop status string is set to "clear", the label in the pMarineViewer window just doesn't show it. An absent all-stop status message implies the all-stop status is "clear".

5.4 Parameters for the pHelmIvP MOOS Configuration Block

The following configuration parameters are defined for the IvP Helm. The parameter names are case insensitive.

Parameter	Mandatory	Description
allow_park	NO	If false (default is true) helm cannot be put into PARK.
behaviors	NO	The name and location of the behavior configuration file.
behavior_dir	NO	A directory to look for dynamically loaded behaviors.
community	YES	Global MOOS parameter. Determines ownship name.
park_on_allstop	NO	If true (default is false) helm will park on all-stop.
domain	YES	The decision space for the IvP Solver.
ok_skew	NO	Tolerance on the age of incoming mail before rejected as being too old.
other_override_var	NO	Names an additional MOOS Variable acting as MOOS_MANUAL_OVERIDE.
start_in_drive	NO	Determines whether or not the helm is in override mode at start-up.
verbose	NO	Determines verbosity of terminal output - quiet, terse, or verbose.

Table 2: Configuration parameters for the pHelmIvP block in a typical MOOS mission configuration file.

The allow_park Parameter

Optional. By setting this parameter to `false`, the helm cannot be manually overriden (parked) once it has been put into `DRIVE`. This can be dangerous and should be carefully considered, and thus the default is `true`. This option was implemented based on experiences with launching UUV autonomy missions and preventing an inadvertent park due to a remote login to the vehicle. There was a tendency for some users to use `iRemote` upon remote login to interact with MOOS, and `iRemote` posts `MOOS_MANUAL_OVERIDE =true` upon launch and connection to the `MOOSDB`.

The behaviors Parameter

Optional (sort of). The parameter names the behavior file, i.e., `*.bhv` file, on the local file system from which the helm behaviors are read. More than one file may be specified on separate lines, and the helm will read in all files almost as if they were one single file. This is technically an optional parameter because a behvior file could be provide on the command line. A behavior file must be specified via one means or the other. If a behavior file is specified *both* on the command line and in the `pHelmIvP` configuration block with this parameter, they will both be used to configure the helm behaviors.

The behavior_dir Parameter

Optional. The parameter names a directory in the local files system where the helm is to look for dynamically loadable behaviors (as opposed to default set built in statically to the helm). Authors augmenting the helm with their own behaviors will need to specify the location of those behaviors with this parameter. More than one line may be provided, each specifying a different directory location.

The community Parameter

This parameter is defined at the "global" level outside of any MOOS process' configuration block. See Section 3.5. The helm reads this parameter and uses its value as the name associated with "ownship". It is a mandatory parameter.

The park_on_allstop Parameter

Optional. By setting this parameter to `true`, the helm will park when an all-stop event occurs. The default setting is `false`.

The domain Parameter

Mandatory. This parameter prescribes the decision space of the helm. It consists of one line per decision variable. Each line contains a colon-separated list of four fields. Field one is the domain variable name, field two is the lower bound value, field three is the higher bound value, and field four is the number of points in the domain. For example `domain = speed:0:3:16` shown in Listing 8 indicates a domain variable called "speed", with a lower and upper bound 0 and 3 meters/second respectively. Since there are 16 points, the speed choices are 0, 0.2, 0.4, ..., 2.8, 3.0. The helm requires that a decision be made on all listed variables on each iteration of the control loop. If a

variable is used by some behaviors but is not necessarily involved in all decisions, it can be declared as optional. For example `domain=speed:0:3:16:optional`.

The `ok_skew` Parameter

Optional. This parameter sets the allowable skew tolerated by the helm for receiving incoming mail messages. If a clock skew is detected greater than this value, the message will be ignored. A check for skews can be disabled by setting `OK_SKEW = ANY`. The default value is 60 seconds.

The `other_override_var` Parameter

Optional. This parameter names a MOOS variable the helm will regard as being synonomous with the two default variables accepted for manual override, `MOOS_MANUAL_OVERRIDE`, and the legacy mispelling of this variable, `MOOS_MANUAL_OVERIDE`.

The `start_in_drive` Parameter

Optional. This parameter is set to either `true` or `false`. The default is `false` as the helm normally starts in `PARK` and needs to receive MOOS mail on the variable `MOOS_MANUAL_OVERIDE` with the value of this variable set to `false`. When `start_in_drive` is set to `true`, the helm is in the `DRIVE` state upon start-up. The issue of helm state was discussed in more detail in Section 5.2.

The `verbose` Parameter

Optional. This parameter affects how much information is written to the terminal on each iteration of the helm. The possible values are *verbose*, *terse*, or *quiet*. The *verbose* setting will write a brief helm report to the terminal on each iteration. With the *terse* setting minimal output will be produced, a '*' character when not producing helm commands, and a '$' character when active and healthy. With the *quiet* setting, no output at all will be written to the terminal. The default value is *terse*. This setting can be changed after the helm is started by changing the value of `HELM_VERBOSE` in the MOOSDB.

An Example pHelmIvP MOOS Configuration Block

Below is an example configuration block for the IvP Helm.

Listing 8 - An example `pHelmIvP` *configuration block.*

```
 0 //-------- pHelmIvP configuration block  -------------
 1 ProcessConfig = pHelmIvP
 2 {
 3   AppTick    = 4   // Defined for all MOOS processes
 4   CommsTick  = 4   // Defined for all MOOS processes
 5
 6   domain     = course:0:359:360
 7   domain     = speed:0:3:16
 8   domain     = depth:0:500:101
 9
10   behaviors  = foobar.bhv
11   verbose    = terse
12   ok_skew    = ANY
13
```

```
14   start_in_drive  = false
15   allow_park      = true
16   park_on_allstop = false
17 }
```

The AppTick and CommsTick parameters are defined for all MOOS processes (see [14]) and specify the frequency in which the helm process iterates and communicates with the MOOSDB. The community parameter is not included in the configuration block because it is specified at the global level in the mission file.

5.5 Launching the IvP Helm and Output to the Terminal Window

The IvP Helm can be launched either directly from the command line, or from within Antler. On the command line the usage is as follows:

```
Usage: pHelmIvP file.moos [file.bhv]...[file.bhv]
       [--help|-h] [--version|-v]

[file.moos] Filename to get MOOS config parameters.
[file.bhv]  Filename to get IvP Helm config paramters.
[-v]        Output version number and exit.
[-h]        Output this usage information and exit.
```

If no behavior file is specified in the .moos file then a behavior file must be given on the command line. Multiple behavior files may be provided. Order of the arguments do not matter - command line arguments ending in .bhv will be read as behavior files, and those ending with .moos as MOOS files. The specification of behavior files may also be split between references in the .moos file and the command line. The duplicate specification of a single file will simply be ignored. Typical start-up output to the terminal is shown in Listing 9 below.

Listing 9 - Example start-up output generated by the pHelmIvP process.

```
 0  ***************************************************
 1  *                                                 *
 2  *       This is MOOS Client                       *
 3  *       c. P Newman 2001                          *
 4  *                                                 *
 5  ***************************************************
 6
 7  ---------------MOOS CONNECT-----------------------
 8    contacting a MOOS server localhost:9000 -  try 00001
 9    Contact Made
10    Handshaking as "pHelmIvP"
11    Handshaking Complete
12    Invoking User OnConnect() callback...ok
13  -------------------------------------------------
14
15  The IvP Helm (pHelmIvP) is starting....
16  Loading behavior dynamic libraries....
17      Loading directory: /Users/mikerb/project-colregs/src/lib_behaviors-colregs
18  Loading behavior dynamic libraries - FINISHED.
19  Number of behavior files: 1
20  Processing Behavior File: bravo.bhv  START
21      Successfully found file: bravo.bhv
22      InitializeBehavior: found static behavior BHV_Loiter
23      InitializeBehavior: found static behavior BHV_Loiter
```

```
24      InitializeBehavior: found static behavior BHV_Waypoint
25      InitializeBehavior: found static behavior BHV_Timer
26  Processing Behavior File: bravo.bhv  END
27  pHelmIvP is Running:
28    AppTick   @ 4.0 Hz
29    CommsTick @ 4 Hz
30    Time Warp @ 1.0
31  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

The output in lines 0-13 are standard output generated by a MOOS process launched and successfully connected to a running MOOSDB. Lines 15-30 are start-up output generated unique to the IvP Helm and the particular user usage. Behaviors used by the helm are either static or dynamic. Static behaviors are compiled in to the pHelmIvP executable. Dynamic behaviors are brought in at run time via shared libraries compiled separately. The helm looks for an environment variable `IVP_BEHAVIOR_DIRS` for a colon-separated list of directories to search for shared libraries. If this variable is not set, or if one or more of the directories are not legitimate directories, an error message will indicate so between what is otherwise line 16 and 18 in Listing 9. This kind of error may not actually be problematic if the behaviors specified in the behavior file can all be otherwise successfully found.

For each specified behavior file, the information shown in lines 20-26 is generated to the terminal. For each behavior configuration in a given `.bhv` file, a single line is output as in lines 22-25 indicating that the behavior type is recognized and it is configured properly. A single unrecognized behavior or improper configuration will result in (a) an error message indicating the offending line number and file name, (b) the output of the actual offending line, and (c) immediate disconnection of the process from the MOOSDB and exit. (Tip: If the helm is launched with Antler an error during start-up will result in the closing of the `pHelmIvP` console window which makes it hard to catch useful error output for debugging. In this case, the helm should just be launched outside of Antler in its own terminal window.)

The output on line 31 of Listing 9, a series of dollar sign characters, indicates for each character, the completion of a single helm iteration - a heartbeat output. This is the output when the `verbose` parameter is set to the default setting of `terse`. When set to `quiet` no output is generated at all. When set to `verbose`, a short multi-line report is generated for each iteration. An example is shown below in Listing 10:

Listing 10 - An example helm iteration report generated by an active helm.

```
0  Iteration: 161  ****************************************
1  Helm Summary  ---------------------------
2  loiter_a did NOT produce an obj-function
3  loiter_b produces obj-function - time:0.00 pcs: 9.00000 pwt: 100.00000
4  waypt_return did NOT produce an obj-function
5  loiter_timer did NOT produce an obj-function
6  Number of Objective Functions: 1
7  DESIRED_SPEED:  2.10
8  DESIRED_COURSE:  145.00
9  (End) Iteration: 161  ****************************************
```

On each iteration the Helm Summary indicates which behaviors produced objective functions (lines 2-5), and for those that did, it indicates the CPU time needed to generate the function, the number of pieces in the piecewise linear IvP function, and its priority weight. Following this,

the decision rendered for current iteration is output with one line per decision variable (lines 7-8). This is a very thin summary of what is going on within the helm and it should be noted that the `uHelmScope` tool is a much better suited for monitoring helm activity and debugging. This tool is described later in Section 11.

5.6 Publications and Subscriptions for IvP Helm

The IvP Helm, like any MOOS process, can be specified in terms of its interface to the `MOOSDB`, i.e., what variables it publishes and what variables it subscribes for. It is impossible to provide a complete specification here since the helm is comprised of behaviors, and the means to include any number of third party behaviors. Each behavior is able to post variable-value pairs, published to the `MOOSDB` by the helm on behalf of the behavior at the end of the iteration. Likewise, each behavior may declare to the helm any number of MOOS variables it would like the helm to register for on its behalf. Barring these variables, published and subscribed for by the helm on behalf of individual behaviors, this section addresses the remaining portion of the helm's publish - subscribe interface.

5.6.1 Variables published by the IvP Helm

Variables published by the IvP Helm are summarized below.

- `IVPHELM_SUMMARY`: Produced on each iteration of the helm for consumption by the `uHelmScope` application. See Section 11. It contains information on the current helm iteration regarding the number of IvP functions created, create time, solve time, which behaviors are active, running, idle, and the decision ultimately produced during the iteration. The summary does not include every component in each summary. Components that have not changed in value since the prior summary are dropped from the present summary. This is motivated by the goal to reducing the log file footprint for the helm.
- `IVPHELM_STATEVARS`: Produced periodically by the helm for consumption by the `uHelmScope` application. See Section 11. It contains a comma-separated list of MOOS variables involved in preconditions of any behavior, i.e., variables affecting behavior run states.
- `IVPHELM_DOMAIN`: Produced once by the helm at start-up for consumption by the `uHelmScope` application. See Section 11. It contains the specification of the IvP Domain in use by the helm.
- `IVPHELM_MODESET`: Produced once by the helm at start-up for consumption by the `uHelmScope` application. See Section 11. It contains the specification of the Hierarchical Mode Declarations, if any, in use by the helm.
- `IVPHELM_STATE`: Written by the helm on each iteration of the `pHelmIvP` MOOS application, regardless of whether the helm is in the `DRIVE` state or not. (see Section 5.2). It is either `"DRIVE"` or `"PARK"`. This is the recommended MOOS variable for regarding as a "heartbeat" indicator of the helm.
- `HELM_IPF_COUNT`: Produced on each iteration of the helm. It contains the number of IvP functions involved in the solver on the current iteration.
- `CREATE_CPU`: The CPU time in seconds used in total by all behaviors on the current iteration for constructing IvP functions.

- LOOP_CPU: The CPU time in seconds used by the IvP solver in the current helm iteration.
- BHV_IPF: The helm will publish this variable for each active behavior in the current iteration. It contains a string representation of the IvP function produced by the behavior. It is used for visualization by the uFunctionVis application, and for logging and later playback and analysis.
- PLOGGER_CMD: This variable is published with the below value to ensure that the pLogger application logs the .bhv file along with the other data log files and the .moos file.

```
"COPY_FILE_REQUEST = filename.bhv"
```

- DESIRED_*: Each of the decision variables in the IvPDomain provided in the helm configuration will have a separate posting prefixed by DESIRED_ as in DESIRED_SPEED. One exception is that the variable course will be converted to heading for legacy reasons.
- BHV_WARNING: Although this variable may never be posted, it is the default MOOS variable used when a behavior posts a warning. A warning may be harmless but deserves consideration.
- BHV_ERROR: Although this variable may never be posted, it is the default MOOS variable used when a behavior posts what it considers a fatal error - one that the helm will interpret as a request to generate the equivalent of ALL-STOP.

In addition to the above variables, the helm will post any variable-value pair on behalf of a behavior that makes the request. These include endflags, runflags, idleflag, activeflags and inactiveflags.

5.6.2 Variables Subscribed for by the IvP Helm

Variables subscribed for by the IvP Helm are summarized below:

- MOOS_MANUAL_OVERIDE: When set to true, usually by a third-party application such as iRemote, of from a command-and-control communication, the helm may relinquish control. If the helm was configured with active_start = true, it will not relinquish control (this may be changed).
- HELM_VERBOSE: Affects the console output produced by the helm. Legal values are verbose, terse, or quiet. See Section 5.5.
- HELM_MAP_CLEAR: When received, the helm clears an internal map that is used to surpress repeated duplicate postings. See Section 5.8.

In addition to the above variables, the helm will subscribe for any variable-value pair on behalf of a behavior that makes the request. This includes, but is not limited to, variables involved in the condition and updates parameters available generally for all behaviors.

5.7 Using a Standby Helm

A *standby helm* refers to the launching of a second helm running alongside another otherwise normally-configured primary helm. This is done to mitigate the risk associated with the possible failure of the primary helm. Both helms are instances of the pHelmIvP MOOS application configured with a different mission and/or behaviors. Presumably the standby helm is configured with a simpler, more conservative set of behaviors focused on the safe recovery of a vehicle. Although

provisions are generally made during vehicle operations to detect a missing helm heartbeat, in situations such as the operation of a UUV under ice, it is not acceptable to simply have a vehicle halt and come to the surface. An attempt should be made to execute a simpler mission to return the vehicle to a safe location for recovery.

Use of a standby helm is as simple as adding a second configuration block in the `.moos` configuration file. The Kilo mission in Section 10.5 on page 174 demonstrates a mission using the standby helm. Declaring a second helm as a standby helm requires a single additional line of the form

```
STANDBY = N
```

where `N` is the number of seconds a standby helm will tolerate an absent primary helm heartbeat before taking control from away the primary helm. See Listing 1 on page 175 for an example. *Once a standby helm take-over is triggered, the take-over is irreversible.*

5.7.1 Two Types of Helm Failure, the Causes, and Detection

What does a helm crash mean, and how might it happen? A crash is result of code that causes the process to quit unexpectedly and without warning. A line as simple as `assert(0);` in the code would be sufficient to replicate a crash, but the cause of a crash could be much more subtle due referencing memory not properly allocated and so on. The other type of helm failure is a helm that *hangs*. This refers to the scenario where the helm enters a piece of code that takes too long or never finishes execution. This could be as trivial as reaching the line `while(1);` somewhere in the code. Either type of failure is serious. Despite the fact that we have never experienced these failures in any field exercise on any vehicle, the sudden disappearance of the helm process should be considered and handled as gracefully as possible.

How is a helm failure detected? The helm produces a heartbeat message on each iteration by posting to the variable `IVPHELM_STATE`. If the helm is configured to iterate four times per second, suspicion of failure could begin anytime after a quarter of a second passes without a posting to this variable. Under typical helm CPU load with common standard behaviors, the quarter second interval should be sufficient to finish the iteration. There are additional factors to consider however. There may be other processes on the machine dominating the CPU, thus challenging the helm to do its work in the expected time. There may be a periodic behavior calculation, such as recalculating a long path of waypoints, that may cause a spike in CPU cycles needed by the behavior. As a rule of thumb, the interval of time with the absence of a heartbeat, should arguably be two seconds or more before declaring a helm failure. This interval is directly configurable, as the parameter `N`, in the `standby=N` configuration line.

5.7.2 Handling a Helm Crash with the Standby Helm

A helm *crash* is the easier of the two failure cases to handle. The process is simply gone and no longer publishes to the MOOSDB. Of course the other MOOS processes, including the standby helm, have no way of knowing simply through their MOOS mailbox whether or not the primary helm is gone or just delayed. In either case the sequence of events is the following:

1. The standby helm detects the absence of heartbeat for more than `N` seconds.

2. On the very same iteration, the standby helm posts IVPHELM_STATE = DRIVE+ rather than IVPHELM_STATE = STANDBY which it had been posting on every iteration up until now. It also begins posting a desired helm decision on this iteration.

The standby helm has all the same functionality as the primary helm, modulo the behavior and mission configuration. It will be in the DRIVE state only if not manually overridden. If MOOS_MANUAL_OVERRIDE=true when the standby helm takes over, it will be in initially in PARK, not DRIVE. It will post IVPHELM_STATE=PARK+. Note the standby helm will append the '+' character to the helm state string to help other applications and the user discern that the helm state being posted is from a standby helm that has taken control.

The Kilo example mission in Section 10.5 walks through a simulated helm crash. The output of the helm(s) prior to and after the standby takeover is discussed in Figure 56.

5.7.3 Handling a Hung Helm with the Standby Helm

Handling a helm that has *hung* requires a bit more consideration. The tricky part is that quite possibly the hung helm is only *temporarily* hung, and at some point it will become unhung and may operate for single iteration as if it is still the helm in charge. Two helms, with different missions, both thinking they are in charge! The sequence of events is summarized below:

1. The standby helm detects the absence of heartbeat for more than N seconds.
2. On the very same iteration, the standby helm posts IVPHELM_STATE = DRIVE+ rather than IVPHELM_STATE = STANDBY which it had been posting on every iteration up until now. It also begins posting a desired helm decision on this iteration.
3. Some time later, the original primary helm finishes its iteration and posts a helm decision, e.g., a set of postings to the helm decision variables, DESIRED_HEADING etc.
4. On the next iteration of the original primary helm it notices that another helm has posted a non-standby heartbeat, e.g., a posting to IVPHELM_STATE not equal to "STANDBY".
5. The original primary helm posts an all-stop and IVPHELM_STATE = DISABLED. It is the last time it will post a heartbeat or helm decision.
6. The standby helm continues to be in control posting helm decisions oblivious to the former primary helm's epiphany and temporary influence.

The key issue in this scenario is that the original primary helm does indeed post a helm decision when it becomes un-hung even though the standby helm may have long ago taken over and may be posting a sequence of helm decisions completely at odds with the decision posted by the newly awakened un-hung primary helm.

No assumptions can be made about the MOOS process listening to the sequence of helm decisions. It may be payload interface process, or a native PID controller, or some other process responsible for converting helm decisions to lower level actuator commands. The assumption that *is* made here is that a one-time aberation in the sequence in helm decisions is tolerable. This aberation is not going to result in an actuator breaking due to a sudden change in the command sequence such as high-speed, zero-speed, high-speed.

It's also worth noting that original primary helm, upon awakening and learning it is no longer in control, does indeed post one more helm decision, an all-stop decision (DESIRED_SPEED = 0). This is done to ensure that an all-stop decision, that may have been put into effect by the standby helm, is not superceded by the output of the original primary helm's final helm decision made upon wakeup.

5.7.4 Activity of the Standby Helm While Standing By

In the standby state the helm will do nothing in its iterate loop other than post a heartbeat character to the terminal if configured to do so. It will not call on any behaviors to do anything. The helm will however read and process all of its otherwise subscribed for mail. In particular it will monitor for postings to IVPHELM_STATE, and will initiate a take-over when it hasn't received such mail for N seconds. Note the standby helm also publishes to this variable, IVPHELM_STATE = STANDBY, but ignores mail originating from itself.

The helm will also read and process all other mail in its inbox, updating the helm's information buffer. The helm's information buffer is described in detail in Section 7.5.2. Note that the information buffer also keeps a *history* of postings to a particular variable so that normally a behavior may process multiple postings if multiple postings were made to the MOOSDB between helm iterations. This history is cleared by the helm at the end of each helm iteration. When the helm is in the standby state it will also clear the history after each iteration even though the behaviors have not been given the chance to access this history. This is to ensure that the information buffer history doesn't grow without bound while in standby mode. For example, if the standby and primary helm are both configured with a waypoint behavior and the primary helm visits half the points at the point of a helm crash, the standby helm's waypoint behavior would start with the first waypoint.

5.7.5 Activity of the Primary Helm After Take-Over

A primary helm that has been taken over is refered to as a *didsabled* helm. It will post only once IVPHELM_STATE = DISABLED. The helm process lives on however doing next to nothing. It does not read its mail, and the iterate loop simply posts a heartbeat character ('!') to the terminal if configured to do so, with the helm configuration parameter verbose=terse.

5.8 Automated Filtering of Successive Duplicate Helm Publications

The helm implements a "duplication filter" to drastically reduce the amount of mail posted by the helm on behalf of behaviors. This filter has been noted to reduce the overall log file size seen during in-water exercises by 60-80%. Reductions at this level noticably facilitate the use of post-mission analysis tools and data archiving. For the most part this filter is operating behind the scenes for the typical helm user. However, knowledge of it is indeed relevant for users wishing to implement their own behaviors, and we discusss it here to explain a bit what is behind the variable HELM_MAP_CLEAR to which the helm subscribes, and listed above in Section 5.6.2.

5.8.1 Motivation for the Duplication Filter

The primary motivation of implementing the duplication filter is to reduce the amount of unnecessary mail posted by the helm on behalf behaviors, and thereby greatly reduce the size of log files

and facilitate the post-mission handling of data. By unnecessary we mean successive variable-value pairs that match exactly in both fields. Surely there are cases when a behavior developer may not want this filter, and there are simple ways to bypass the filter for any post. But in most cases, successive duplicate posts are just redundant and unnecessary.

5.8.2 Implementation and Usage of the Duplication Filter

The helm keeps two maps (STL maps in C++), one for string data and one for numerical data:

```
KEY --> StringValue
KEY --> DoubleValue
```

The two maps correspond to the double and string message types in MOOS (see Section 3.2 on page 22). The `KEY` is typically the MOOS variable name. Inside a behavior implementation, the following four functions are available:

```
void postMessage(string varname, string value, string key="");
void postMessage(string varname, double value, string key="");
void postBoolMessage(string varname, bool value, string key="");
void postIntMessage(string varname, double value, string key="");
```

These functions are available in all behavior implementations because they are defined in the IvPBehavior superclass, of which all behaviors are subclasses. Before the helm posts a message to the `MOOSDB` the filter is applied by a simple check to its map to determine if there is a value match on the given key. If a match is made, the post will not be made to the `MOOSDB` on the behavior's behalf. The `postIntMessage()` function is merely a convenience version of the `postMessage()` function that rounds the variable value to the nearest integer to further reduce posts when combined with the filter. The `postBoolMessage()` ultimately posts a string value `"true"` or `"false"`.

The default value of the *key* parameter is the empty string, and in most cases this parameter can be ommitted without disabling the duplication filter. This is because the `KEY` used by the caller is only part of the key actually used by the duplication filter. The actual key is the concatenation of (a) the behavior name, (b) the variable name, and (c) the key passed by the caller. Thus the default value, the empty string, still results in a decent key being used by the filter. The key is augmented by the behavior name because often there is more than one behavior posting messages on same variable. The optional key parameter is used for two reasons. First, it can be used to further distinguish posts within a behavior on the same variable name. Second, when the key value has the special value `"repeatable"`, then no key is used and the duplication filter is disabled for that variable posting.

5.8.3 Clearing the Duplication Filter

Occasionally a user, or another MOOS application in the same community as the helm, may want to "clear" the map used by the helm to implement its duplication filter. This can be done by writing to variable `HELM_MAP_CLEAR`, with any value. This may be necessary for the following reason. Suppose a GUI application subscribes for the variable `VIEW_SEGLIST` which contains a list of line segments for rendering. If the viewer application is launched *after* the variable is published, the application will only receive the most recent mail on the variable `VIEW_SEGLIST`. There may be publications to

this variable, made prior to the most recent publication, that are relevant to the GUI application at launch time. Those publications for the variable VIEW_SEGLIST may not be the most recent from the perspective of the MOOSDB, but they may be the most recent from the perspective of a particular behavior in the helm. By clearing the filter, it gives each behavior the chance to once again have all of its variable-value posts made to the MOOSDB. In the pMarineViewer application, a publication to HELM_MAP_CLEAR is made upon start-up. Clearing the filter will only clear the way for the next post for a given variable. It will not result in the publishing to the MOOSDB of the contents of the maps used by the filter.

6 IvP Helm Autonomy

6.1 Overview

An autonomous helm is primarily an engine for decision making. The IvP Helm uses a behavior-based architecture to organize its decision making and is distinctive in the manner in which it resolves competition between competing behaviors - it performs multi-objective optimization on their collective output using a mathematical programming model called interval programming. Here the IvP Helm architecture is described and the means for configuring it given a set of behaviors and a set of mission objectives.

6.1.1 The Influence of Brooks, Stallman and Dantzig on the IvP Helm

The notion of a behavior-based architecture for implementing autonomy on a robot or unmanned vehicle is most often attributed to Rodney Brooks' Subsumption Architecture, [8]. A key principle at the heart of Brooks' architecture and arguably the primary reason its appeal has endured, is the notion that autonomy systems can be built *incrementally*. Notably, Brooks' original publication pre-dated the arrival of Open Source software and the Free Software Foundation founded by Richard Stallman. Open Source software is not a pre-requisite for building autonomy systems incrementally, but it has the capability of greatly accelerating that objective. The development of complex autonomy systems stands to significantly benefit if the set of developers at the table is large and diverse. Even more so if they can be from different organizations with perhaps even the loosest of overlap in interest regarding how to use the collective end product.

As discussed in Section 2.5, a key issue in behavior-based autonomy has been the issue of action selection, and the IvP Helm is distinct in this regard with the use of multi-objective optimization and interval programming. The algorithm behind interval programming, as well as the term itself, was motivated by the mathematical programming model, linear programming, developed by George Dantzig, [10]. The key idea in linear programming is the choice of the particular mathematical construct that comprises an instance of a linear programming problem - it has enough expressive flexibility to represent a huge class of practical problems, *and* the constructs can be effectively exploited by the simplex method to converge quickly even on very large problem instances. The constructs used in interval programming to represent behavior output (piecewise linear functions) were likewise chosen to have enough expressive flexibility to handle any current and future behavior, and due to the opportunity to develop solution algorithms that exploit the piecewise linear constructs.

6.1.2 Traditional and Non-traditional Aspects of the IvP Behavior-Based Helm

The IvP Helm indeed takes its motivation from early notions of the behavior-based architecture, but is also quite different in many regards. The notion of behavior independence to temper the growth of complexity in progressively larger systems is still a principle closely followed in the IvP Helm. Behaviors may certainly influence one another from one iteration to the next, as we'll see in discussions in this section. This was also evident in the Alpha example mission in Section 4 where the completion of the Survey behavior triggered the Return behavior. But within a single iteration, the output generated by a single behavior is not affected at all by what is generated by other behaviors in the same iteration. The only inter-behavior "communication" realized within an

iteration comes when the IvP solver reconciles the output of multiple behaviors. The independence of behaviors not only helps a single developer manage the growth of complexity, but it also limits the dependency between developers. A behavior author need not worry that a change in the implementation of another behavior by another author requires subsequent recoding of one's own behavior(s).

Certain aspects of behaviors in the IvP Helm may also be a departure from some notions traditionally associated (fairly or not) with behavior-based architectures:

- Behaviors have state. IvP behaviors are instances of a class with a fairly simple interface to the helm. Inside they may be arbitrarily complex, keep histories of observed sensor data, and may contain algorithms that could be considered "reactive" or "plan-based".
- Behaviors influence each other between iterations. The primary output of behaviors is their objective function, ranking the utility of candidate actions. IvP behaviors may also generate variable-value posts to the `MOOSDB` observable by behaviors on the next helm iteration. In this way they can explicitly influence other behaviors by triggering or suppressing their activation or even affecting the parameter configuration of other behaviors.
- Behaviors may accept externally generated plans. The input to a behavior can be anything represented by a MOOS variable, and perhaps generated by other MOOS processes outside the helm. It is allowable to have one or more planning engines running on the vehicle generating output consumed by one or more behaviors.
- Several instances of the same behavior. Behaviors generally accept a set of configuration parameters that allow them to be configured for quite different tasks or roles in the same helm and mission. Different waypoint behaviors, for example, can be configured for different components of a transit mission. Or different collision avoidance behaviors can be instantiated for different contacts.
- Behaviors can be run in a configurable sequence. Due to the `condition` and `endflag` parameters defined for all behaviors, a sequence of behaviors can be readily configured into a larger mission plan.
- Behaviors rate actions over a coupled decision space. IvP functions generated by behaviors are defined over the Cartesian product of the set of vehicle decision variables. This is distinct from the de-coupled decision making style proposed in [15] and [17] - early advocates of multi-objective optimization in behavior-based action selection.

6.1.3 Two Layers of Building Autonomy in the IvP Helm

The autonomy in play on a vehicle during a particular mission is the product of two distinct efforts - (1) the development of vehicle behaviors and their algorithms, and (2) mission planning via the configuration of behaviors and mode declarations. The former involves the writing of new source code, and the latter involves the editing of mission behavior files, such as the simple example for the Alpha example mission in Listing 5 on page 40.

6.2 Inside the IvP Helm - A Look at the Helm Iterate Loop

Like other MOOS applications, the IvP Helm implements an `Iterate()` loop within which the basic function of the helm is executed. Components of the `Iterate()` loop, with respect to the behavior-based architecture, are described in this section. The basic flow, in five steps, is depicted in Figure 12. Description of the five components follow.

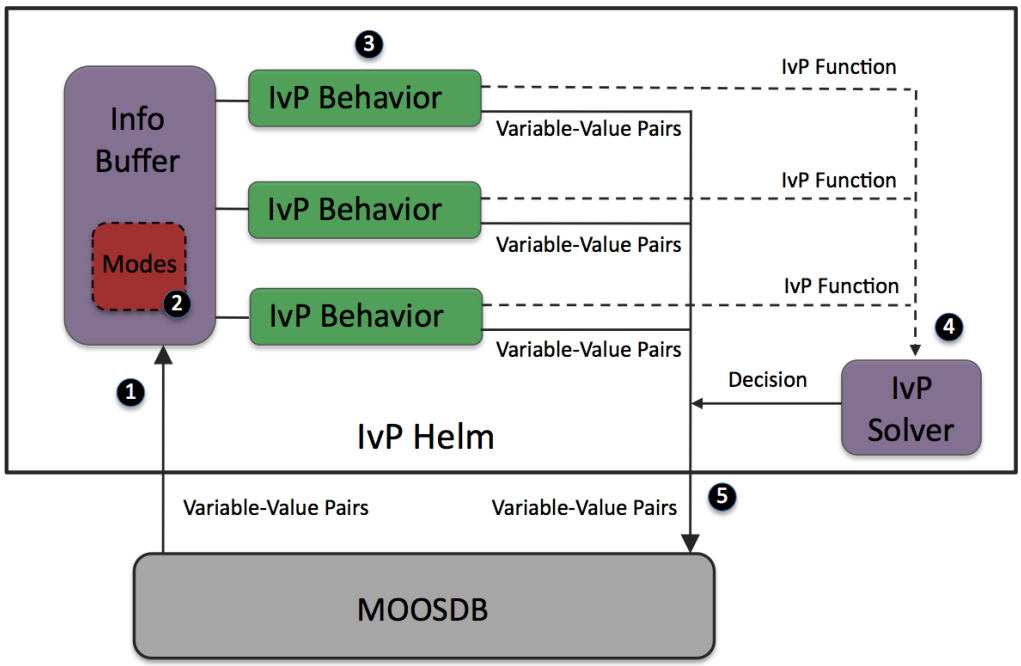


Figure 12: **The pHelmIvP Iterate Loop**: (1) Mail is read from the MOOSDB. It is parsed and stored in a local buffer to be available to the behaviors, (2) If there were any mode declarations in the mission behavior file they are evaluated at this step. (3) Each behavior is queried for its contribution and may produce an IvP function and a list of variable-value pairs to be posted to the MOOSDB at the end of the iteration, (4) the objective functions are resolved to produce an action, expressible as a set of variable-value pairs, (5) all variable-value pairs are published to the MOOSDB for other MOOS processes to consume.

6.2.1 Step 1 - Reading Mail and Populating the Info Buffer

The first step of a helm iteration occurs outside the `Iterate()` loop. As depicted in Figure 5 on page 25, a MOOS application will read its mail by executing its `OnNewMail()` function just prior to executing its `Iterate()` loop if there is any mail in its in-box. The helm parses mail to maintain its own information buffer which is also a mapping of variables to values. This is done primarily for simplicity - to ensure that each behavior is acting on the same world state as represented by the info buffer. Each behavior has a pointer to the buffer and is able to query the current value of any variable in the buffer, or get a list of variable-value changes since the previous iteration.

6.2.2 Step 2 - Evaluation of Mode Declarations

Once the information buffer is updated with all incoming mail, the helm evaluates any mode declarations specified in the behavior file. Mode declarations are discussed in Section 6.4. In short, a mode is represented by a string variable that is reset on each iteration based on the evaluation of a set of logic expressions involving other variables in the buffer. The variable representing the mode declaration is then available to the behavior on the current iteration when it, for example, evaluates its `condition` parameters. A condition for behavior participating in the current iteration could therefore read something like `condition = (MODE==SURVEYING)`. The exact value of the variable `MODE` is set during this step of the `Iterate()` loop.

6.2.3 Step 3 - Behavior Participation

In the third step much of the work of the helm is realized by giving each behavior a chance to participate. Each behavior is queried sequentially - the helm contains no separate threads in this regard. The order in which behaviors is queried does not affect the output. This step contains two distinct parts for each behavior - (1) Determination of whether the behavior will participate, and (2) production of output if it is indeed participating on this iteration. Each behavior may produce two types of information as the Figure 12 indicates. The first is an objective function (or "utility" function) in the form of an IvP function. The second kind of behavior output is a list of variable-value pairs to be posted by the helm to the MOOSDB at the end of the `Iterate()` loop. A behavior may produce both kinds of information, neither, or one or the other, on any given iteration.

6.2.4 Step 4 - Behavior Reconciliation

In the fourth step depicted in Figure 12, the IvP functions are collected by the IvP solver to produce a single decision over the helm's decision space. Each function is an IvP function - an objective function that maps each element of the helm's decision space to a utility value. In this case the functions are of a particular form - piecewise linearly defined. That is, each piece is an *interval* of the decision space with an associated linear function. Each function also has an associated weight and the solver performs multi-objective optimization over the weighted sum of functions (in effect a single objective optimization at that point). The output is a single optimal point in the decision space. For each decision variable the helm produces another variable-value pair, such as `DESIRED_SPEED = 2.4` for publication to the MOOSDB.

6.2.5 Step 5 - Publishing the Results to the MOOSDB

In the last step, the helm simply publishes all variable-value pairs to the MOOSDB, some of which were produced directly by the behaviors, and some of which were generated as output from the IvP Solver. The helm employs the duplication filter described in Section 5.8, only on the variable-value pairs generated directly from the behaviors, and not the variable-value pairs generated by the IvP solver that represent a decision in the helm's domain. For example, even if the decision about a vehicle's depth, represented by the variable `DESIRED_DEPTH` produced by the helm were unchanged for 5 minutes of operation, it would be published on each iteration of the helm. To do otherwise could give the impression to consumers of the variable that the variable is "stale", which could trigger an unwanted override of the helm out of concern for safety.

6.3 Mission Behavior Files

The helm is configured for a particular mission primarily through one or more mission behavior files, typically with a `*.bhv` suffix. Behavior files have three types of entries, usually but not necessarily kept in three distinct parts - (1) variable initializations, (2) behavior configurations, and (3) hierarchical mode declarations. These three parts are discussed below. The example `alpha.bhv` file in Listing 5 on page 40 did not contain hierarchical mode declarations, but does contain examples of variable initializations and behavior configurations.

6.3.1 Variable Initialization Syntax

The syntax for variable initialization is fairly straight-forward:

```
initialize  <variable> = <value>
...
initialize  <variable> = <value>
```

Multiple initializations may be declared on a single line by separating each variable-value pair with a comma. The keyword `initialize` is case insensitive. The `<variable>` is indeed case sensitive since it will be published to the `MOOSDB` and MOOS variables are case sensitive when registered for by a client. The `<value>` may or may not be case sensitive depending on whether or not a client registering for the variable regards the case. Considering again the helm `Iterate()` loop depicted in Figure 12 on page 63, variable initializations are applied to the helm's information buffer prior to the very first helm iteration, but are posted to the `MOOSDB` at the end of the first helm iteration.

By default, an initialization will overwrite any prior value posted to the MOOSDB. There may be situations, however, where the user's desired effect is that the initialization only be applied if no other value has yet been written to the given MOOS variable. The syntax in this case would be:

```
initialize_  <variable> = <value>    // Deferring to prior posts if any
```

By using the "underscore" version of the `initialize` declaration, the helm will first register with the MOOSDB for the given variable, wait an iteration until it has had chance to receive mail from the MOOSDB on that variable, and only initialize the variable is nothing is known otherwise about that variable. (Note to the very discerning reader: Such an initialization also includes both an update to the helm's information buffer and a post to the MOOSDB. Posts to the MOOSDB by the helm, as part of a variable initialization, will indeed show up in the helm's incoming mailbox on the next iteration, but they are tagged in such a way as to be ignored by the helm. This is to ensure that they do not "collide" with posts made by other processes.)

6.3.2 Behavior Configuration Syntax

The bulk of the helm configuration is done with individual behavior parameter blocks which have the following form:

```
Behavior = <behavior-type>
{
  <parameter> = <value>
```

```
  ...
  <parameter> = <value>
}
```

The first line is a declaration of the behavior type. The keyword `Behavior` is not case sensitive, but the `<behavior-type>` is. This is followed by an open brace on a separate line. Each subsequent line sets a particular parameter of the behavior to a given value. The behavior configuration concludes with a close brace on a separate line. The issue of case sensitivity for the `<parameter>` and `<value>` entries is a matter determined by the individual behavior implementation.

As a convention (not enforced in any way) general behavior parameters, defined at the IvP Behavior superclass level, are grouped together and listed before parameters that apply to a specific behavior. For example, in the Alpha example in Listing 5 on page 40, the general behavior parameters are listed on lines 8-12 and 22-25, but the parameters specific to the waypoint behavior, `speed`, `radius`, and `points`, follow in a separate block. Generally it is not mandatory to provide a parameter-value pair for each parameter defined for a behavior, given that meaningful defaults are in place within the behavior implementation. Some parameters are indeed mandatory however. Documentation for the individual behavior should be consulted. Multiple instances of a behavior type are allowed, as in the Alpha example where there are two waypoint behaviors - one for traversing a set of points, and one for returning to a vehicle recovery point. Each behavior should have its own unique value provided in the `name` parameter.

6.3.3 Hierarchical Mode Declaration Syntax

Hierarchical Mode Declarations are covered in depth in Section 6.4, but the syntax is briefly discussed here. A behavior file contains a set of declaration blocks of the form:

```
Set <mode-variable-name> = <mode-value>
{
  <mode-variable-name> = <parent-value>
  <condition>
  . . .
  <condition>
} <else-value>
```

A tree will be formed where each node in the tree is described from the above type of declaration. The keyword `Set` is case insensitive. The `<mode-variable-name>`, `<parent-value>` and `<else-value>` are case sensitive. The `<condition>` entries are treated exactly as with the `condition` parameter for behaviors, see Section 6.5.1.

As indicated in Figure 12, the value of each mode variable is reset at the outset of the `Iterate()` loop, after the information buffer is updated with incoming mail. A mode variable is set by progressing through each declaration block, and determining whether the conditions are met. Thus the ordering of the declaration blocks is significant - the specification of parent should be made prior to that of a child. Examples are further discussion can be found below in Section 6.4.

6.4 Hierarchical Mode Declarations

Hierarchical mode declarations (HMDs) are an optional feature of the IvP Helm for organizing the behavior activations according to declared mission modes. Modes and sub-modes can be declared, in line with a mission planner's own concept of mission evolution, and behaviors can be associated with the declared modes. In more complex missions, it can facilitate mission planning (in terms of less time and better detection of human errors), and it can facilitate the understanding of exactly what is happening in the helm - during the mission execution and in post-analysis.

6.4.1 Background

A trend of unmanned vehicle usage can be characterized as being increasingly less of the shorter, scripted variety to be increasingly more of the longer, adaptive mission variety. A typical mission in our own lab five years ago would contain a certain set of tasks, typically waypoints and ultimately a rendezvous point for recovering the vehicle. Data acquired during deployment was off-loaded and analyzed later in the laboratory. What has changed? The simultaneous maturation of acoustic communications, on-board sensor processing, and longer vehicle battery life has dramatically changed the nature of mission configurations. The vehicle is expected to adapt to both the phenomena it senses and processes on board, as well as adapt its operation given field-control commands received via acoustic, radio or satellite communications. Multi-vehicle collaborative missions are also increasingly viable due to lower vehicle costs and mature acomms capabilities. In such cases a vehicle is not only adapting to sensed phenomena and field commands, but also to information from collaborating vehicles.

Our missions have evolved from having a finite set of fixed tasks to be composed instead of a set of modes, an initial mode when launched, an understanding of what brings us from one mode to another, and what behaviors are in play in each mode. Modes may be entered and exited any number of times, in exact sequences unknown at launch time, depending on what they sense and how they are commanded in the field.

6.4.2 Behavior Configuration *Without* Hierarchical Mode Declarations

Behaviors can be configured for a mission without the use of hierarchical mode declarations - support for HMDs is a relatively recent addition to the helm. HMDs are a tool for organizing which behaviors are idle or participating in which circumstances. Consider the alpha example mission in Section 4, and the behavior file in Listing 5. By examination of the behavior file, and experimenting a bit with the viewer during simulation, the vehicle apparently is always in one of three modes - (a) idle, (b) surveying the waypoints, or (c) returning to the launch point. This is achieved by the `condition` parameters for the two behaviors. There are only two variables involved in the behavior conditions, `DEPLOY` and `RETURN`. If restricted to Boolean values, the below table confirms the observation that there are only three possible modes.

DEPLOY	RETURN	Mode
true	true	Returning
true	false	Surveying
false	true	Idle
false	false	Idle

Table 3: Possible modes implied by the condition parameters in the alpha mission in Listing 5.

There are a couple drawbacks with this however. First, the modes are to be inferred from the behavior conditions and this is not trivial in missions with larger behavior files. Mapping the behavior conditions to a mode is useful both in mission planning and mission monitoring. In the alpha mission, in order to understand at any given moment what mode the vehicle is in, the two variables need to be monitored, and the above table internalized. The second drawback is the increased likelihood of error, in the form of unintentionally being in two modes at the same time, or being in an undefined mode. For example, line 11 in Listing 5 really should read `RETURN != true`, and not `RETURN = false`. Since there is no Boolean type for MOOS variables, this variable could be set to `"False"` and the condition as it reads on line 11 in Listing 5 would not be satisfied, and the vehicle would be in the idle state, despite the fact that `DEPLOY` may be set to `true`. These problems are alleviated by the use of hierarchical mode declarations.

6.4.3 Syntax of Hierarchical Mode Declarations - The Bravo Mission

An example is provided showing of the use of hierarchical mode declarations by extending the Alpha mission described in Section 4. This example mission is dubbed the "Bravo" mission in the directory `s2_bravo` alongside the Alpha mission `s1_alpha` in the MOOS-IvP distribution (Section 4.1). It is also given fully in Listing 11 on the next page. The *implicit* modes of the Alpha mission, described in Table 3, are explicitly declared in the Bravo behavior file to form the following hierarchy:

```
// Excerpt from the
// bravo.bhv file.

Set MODE = Active {
  DEPLOY = true
} Inactive

Set MODE = Surveying {
  MODE = Active
  RETURN != true
} Returning
```

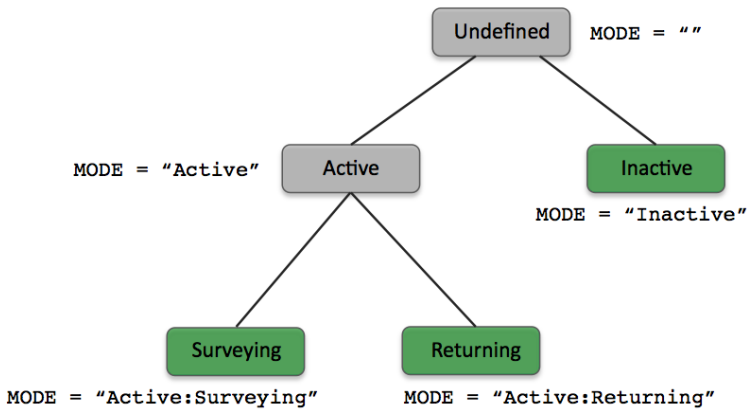


Figure 13: **Hierarchical modes for the Bravo mission**: The vehicle will always be in one of the modes represented by a leaf node. A behavior may be associated with any node in the tree. If a behavior is associated with an internal node, it is also associated with all its children.

The hierarchy in Figure 13 is formed by the mode declaration constructs on the left-hand side, taken as an excerpt from the `bravo.bhv` file. After the mode declarations are read when the helm is initially launched, the hierarchy remains static thereafter. The hierarchy is associated with a particular MOOS variable, in this case the variable `MODE`. Although the hierarchy remains static, the mode is re-evaluated at the outset of each helm iteration based on the conditions associated with nodes in the hierarchy. The mode evaluation is represented as a string in the variable `MODE`. As shown in Figure 13 the variable is the concatenation of the names of all the nodes. The mode evaluation begins sequentially through each of the blocks. At the outset the value of the variable `MODE` is reset to the empty string. After the first block in Figure 13 `MODE` will be set to either `"Active"` or `"Inactive"`. When the second block is evaluated, the condition `"MODE=Active"` is evaluate based on how `MODE` was set in the first block. For this reason, mode declarations of children need to be listed after the declarations of parents in the behavior file.

Once the mode is evaluated, at the outset of the helm iteration, it is available for use in the conditions of the behaviors, as in lines 20 and 23 in Listing 11. Note the `"=="` relation in lines 18 and 36. This is a string-matching relation that matches when one side matches exactly one of the components in the other side's colon-separated list of strings. Thus `"Active" == "Active:Returning"`, and `"Returning" == "Active:Returning"`. This is to allow a behavior to be easily associated with an internal node regardless of its children. For example if a collision-avoidance behavior were to be added to this mission, it could be associated with the `"Active"` mode rather than explicitly naming all the sub-modes of the `"Active"` mode.

Listing 11: The Bravo Mission - Use of Hierarchical Mode Declarations.

```
 0  initialize   DEPLOY = false
 1  initialize   RETURN = false
 2
 3  //------------------- Declaration of Hierarchical Modes
 4  set MODE = ACTIVE {
 5    DEPLOY = true
 6  } INACTIVE
 7
 8  set MODE = SURVEYING {
 9    MODE = ACTIVE
10    RETURN != true
11  } RETURNING
12
13  //---------------------------------------------
14  Behavior = BHV_Waypoint
15  {
16    name      = waypt_survey
17    pwt       = 100
18    condition = MODE == SURVEYING
19    endflag   = RETURN = true
20    perpetual = true
21
22            lead = 8
23     lead_damper = 1
24           speed = 2.0   // meters per second
25          radius = 4.0
26       nm_radius = 10.0
27          points = 60,-40:60,-160:150,-160:180,-100:150,-40
28          repeat = 1
29  }
30
31  //---------------------------------------------
```

```
32  Behavior = BHV_Waypoint
33  {
34    name       = waypt_return
35    pwt        = 100
36    condition  = MODE == RETURNING
37    perpetual  = true
38    endflag    = RETURN = false
39    endflag    = DEPLOY = false
40
41         speed = 2.0
42        radius = 2.0
43     nm_radius = 8.0
44         point = 0,0
45  }
```

6.4.4 A More Complex Example of Hierarchical Mode Declarations

The Bravo example given above, while having the benefit of being a working example distributed with the codebase, is not complex. In this section a modestly complex, although fictional, hierarchy is provided to highlight some issues with the syntax. The hierarchy with the corresponding mode declarations are shown in Figure 14. The declarations are given in the order of layers of the tree ensuring that parents are declared prior to children. As with the Bravo example in Figure 13, the nodes that represent realizable modes are depicted in the darker (green) color.

```
// Level 1 of the Mode Tree
Set MODE = Alpha {
  MISSION = SURVEYING
}
Set MODE = Bravo {
  MISSION = LOITERING
} Charlie

// Level 2 of the Mode Tree
Set MODE = Delta {
  MODE = Alpha
  SITE = Archipelagos
} Echo
Set MODE = Foxtrot {
  MODE = Charlie
  VIDEO = Streaming
} Golf

// Level 3 of the Mode Tree
Set Mode = Sierra {
  MODE = Alpha:Echo
  WATER_DEPTH = Shallow
}
Set Mode = Tango {
  MODE = Alpha:Echo
  WATER_DEPTH = Deep
}
```

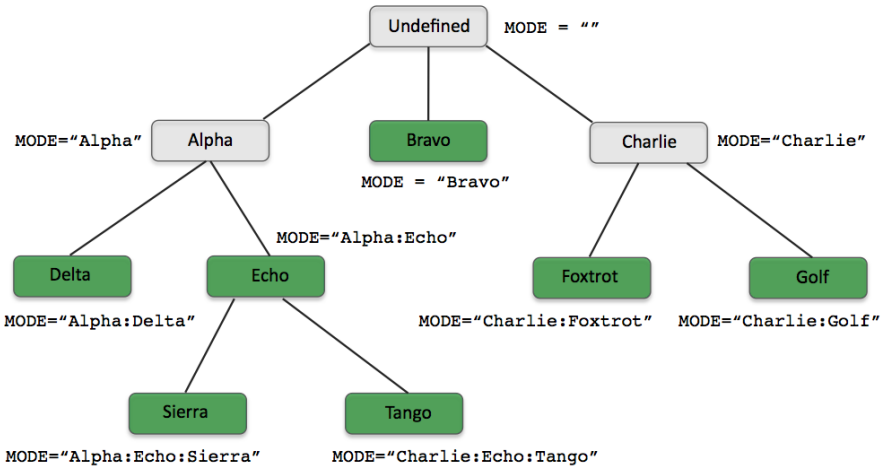


Figure 14: **Example Hierarchical Mode Declaration**: The hierarchy on the right is constructed from the set of mode declarations on the the left (with fictional conditions). Darker nodes represent modes that are realizable through some combination of conditions.

The `"Alpha"` mode for example is not realizable since it has the children `"Delta"` and `"Echo"`, with the latter being set as the `<else-value>` if the conditions of the former at not met. The `"Bravo"` mode is realizable since it has no children. The `"Echo"` mode is realizable despite having children

because the "Tango" mode is not the <else-value> of the "Sierra" mode declaration. For example, if the following three conditions hold, (a) "MISSION=SURVEYING", (b) "SITE!=Archipelagos", and (c) "WATER_DEPTH=Medium", then the value of the variable MODE would be set to "Alpha:Echo". Finally, note that the condition in the "Sierra" declaration, "MODE=Alpha:Echo", is specified fully, i.e., "MODE=Echo" would not achieve the desired result.

6.4.5 Monitoring the Mission Mode at Run Time

The mission mode can be monitored at run time in a couple ways. First, since the mode variable is posted as a MOOS variable, any MOOS scope tool will work, e.g., uXMS, uMS, uHelmScope. Using uHelmScope, the mission variable can be monitored as part of the basic MOOSDB scoping capability (see Section 11.3), but it is also displayed on it's own, in the fourth line of the main output. For example, see line 4 in Listing 28 on page 185.

The uHelmScope tool also has a mode in which the entire mode hierarchy may be rendered - solely to provide a visual confirmation that the hierarchy specified with the mode declarations in the behavior file does in fact correspond to what the user intended. Currently there are no tools to automatically render the mode hierarchy in a manner like the right hand side of Figure 14. The uHelmScope output for the example in Figure 14 is shown in listing 12 below.

Listing 12: The mode hierarchy output from uHelmScope *for the example in Figure 14.*

```
 0   ModeSet Hierarchy:
 1   -----------------------------------------------
 2   Alpha
 3       Delta
 4       Echo
 5           Sierra
 6           Tango
 7   Bravo
 8   Charlie
 9       Foxtrot
10       Golf
11   -----------------------------------------------
12   CURENT MODE(S): Charlie:Foxtrot
13
14   Hit 'r' to resume outputs, or SPACEBAR for a single update
```

More on this feature of the uHelmScope can be found in Section 11. It's worth noting that poking the value of a mode variable will have no effect on the helm operation. The mission mode cannot be commanded directly. The mode variable is reset at the outset of the helm iteration, and the helm doesn't even register for mail on mode variables.

6.5 Behavior Participation in the IvP Helm

The primary work of the helm comes when the behaviors participate and do their thing, at each round of the helm Iterate() loop. As depicted in Figure 12 on page 63, once the mode has been re-evaluated taking into consideration newly received mail, it is time for the behaviors (well, some at least) to step up and do their thing.

6.5.1 Behavior Run Conditions

On any single iteration a behavior may participate by generating an objective function to influence the helm's output over its decision space. Not all behaviors participate in this regard, and the primary criteria for participation is whether or not it has met each of its "run conditions". These are the conditions laid out in the behavior file of the form:

```
condition = <logic-expression>
```

The `<logic-expression>` syntax is described in Appendix A. Conditions are built from simple relational expressions, the comparison of MOOS variables to specified literal values, or the comparison of MOOS variables to one another. Conditions may also involve Boolean logic combinations of relation expressions. A behavior may base its conditions on any MOOS variable such as:

```
condition = (DEPLOY=true) and (STATION_KEEP != true)
```

A run condition may also be expressed in terms of a helm mode, as described in the next Section 6.5.2 such as:

```
condition = (MODE == LOITERING)
```

All MOOS variables involved in run condition expressions are automatically subscribed for by the helm to the MOOSDB.

6.5.2 Behavior Run Conditions and Mode Declarations

The use of hierarchical mode declarations potentially simplify the expressions used as run conditions. The conditions in practice could be limited to:

```
condition = <mode-variable> = <mode-value>, or
condition = <mode-variable> == <mode-value>.
```

Conditions were used in this way with the Bravo mission in Listing 11 on page 69, as an alternative to their usage in the Alpha mission example in Listing 5 on page 40.

Note the use of the double-equals relation above. This relation is used for matching against the strings used to represent the hierarchical mode. The two strings match if the ordered components of one side are a subset of the ordered components of the other. Components are colon-separated. For example, using the illustrative hierarchy from Figure 14:

```
"Alpha:Echo:Sierra" == "Sierra"
"Alpha:Echo:Sierra" == "Echo:Sierra"
"Alpha:Echo:Sierra" == "Alpha"
          "Sierra" == "Alpha:Echo:Sierra"
  "Charlie:Foxtrot" == "Charlie:Foxtrot"

"Alpha:Echo:Sierra" != "Alpha:Sierra"
```

6.5.3 Behavior Run States

On any given helm iteration a behavior may be in one of four states depicted in Figure 15:

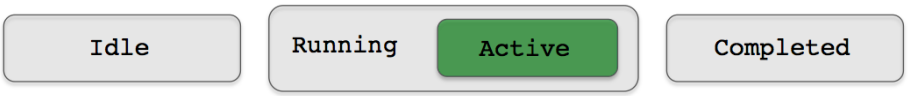


Figure 15: **Behavior States:** A behavior may be in one of these four states at any given iteration of helm `Iterate()` loop. The state is determined by examination of MOOS variables stored locally in the helm's information buffer.

- Idle: A behavior is `idle` if it is not `complete` and it has not met its run conditions as described above in Section 6.5.1. The helm will invoke an idle behavior's `onIdleState()` function.
- Running: A behavior is `running` if it has met its run conditions and it is not `complete`. The helm will invoke a running behavior's `onRunState()` function thereby giving the behavior an opportunity to contribute an objective function.
- Active: A behavior is `active` if it is running and it did indeed produce an objective function when prompted. There are a number of reasons why a running behavior may not be active. For example, a collision avoidance behavior where the object of the behavior is sufficiently far away.
- Complete: A behavior is `complete` when the behavior itself determines it to be complete. It is up to the behavior author to implement this, and some behaviors may never complete. The function `setComplete()` is defined generally at the behavior superclass level, for calling by a behavior author. This provides some some standard steps to be taken upon completion, such as posting of `endflags`, described below in Section 6.5.4. Once a behavior is in the `complete` state, it remains in that state permanently. All behaviors have a `duration` parameter defined to allow it to be configured to time-out if desired. When a time-out occurs the behavior state will be set to `complete`.

6.5.4 Behavior Flags and Behavior Messages

Behaviors may post some number of messages, i.e., variable-value pairs, on any given iteration (see Figure 12, p. 63). These message can be critical for coordinating behaviors with each other and to other MOOS processes. The can also be invaluable for monitoring and debugging behaviors configured for particular missions. To be more accurate, behaviors don't post messages to the `MOOSDB`, they request the helm to post messages on its behalf. The helm collects these requests and publishes them to the `MOOSDB` at the end of the `Iterate()` loop. It also filters them for successive duplicates as discussed in Section 5.8.

There is a standard method, configurable in the behavior file, for posting messages based on the run state of the behavior. These are referred to as behavior flags, and there are five types, (1) `endflag`, (2) `idleflag`, (3) `runflag`, (4) `activeflag`, (5) `inactiveflag`. The variable-value pairs

representing each flag are set in the behavior file for the corresponding behavior. See line 12 in 5 on page 40 for example.

- `endflag`: An `endflag` is posted once when or if the behavior enters the `complete` state. The variable-value pair representing the endflag is given in the `endflag` parameter in the behavior file. Multiple endflags may be configured for a behavior.
- `idleflag`: An `idleflag` is posted by the helm when the behavior enters the `idle` state. The variable-value pair representing the idleflag is given in the `idleflag` parameter in the behavior file. Multiple idleflags may be configured for a behavior.
- `runflag`: An `runflag` is posted by the helm when the behavior enters the `running` state from the `idle` state. A `runflag` is posted exactly when an `idleflag` is not. The variable-value pair representing the runflag is given in the `runflag` parameter in the behavior file. Multiple runflags may be configured for a behavior.
- `activeflag`: An `activeflag` is posted by the helm when the behavior enters the `active` state. The variable-value pair representing the activeflag is given in the `activeflag` parameter in the behavior file. Multiple activeflags may be configured for a behavior.
- `inactiveflag`: An `inactiveflag` is posted by the helm when the behavior enters a state that is not the `active` state. The variable-value pair representing the inactiveflag is given in the `inactiveflag` parameter in the behavior file. Multiple inactiveflags may be configured for a behavior.

A `runflag` is meant to "complement" an `idleflag`, by posting exactly when the other one does not. Similarly with the `inactiveflag` and `activeflag`. The situation is shown in Figure 16:

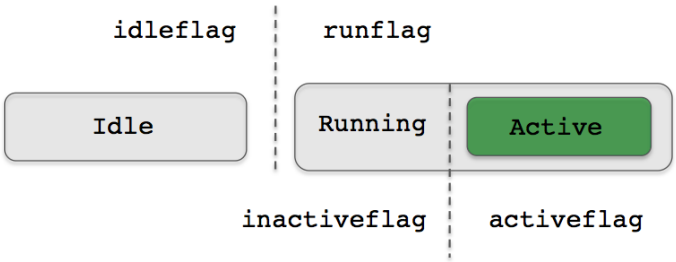


Figure 16: **Behavior Flags:** The four behavior flags `idleflag`, `runflag`, `activeflag`, and `inactiveflag` are posted depending on the behavior state and can be considered complementary in the manner indicated.

Behavior authors may implement their behaviors to post other messages as they see fit. For example the waypoint behavior used in the Alpha example in Section 4 also published the variable `WPT_STAT` with a status message similar to `"vname=alpha,index=0,dist=124,eta=62"` indicating the name of the vehicle, the index of the next point in the list of waypoints, the distance to that waypoint, and the estimated time of arrival, in seconds. (You might want to re-run the Alpha mission with `uXMS` scoping on this variable to watch it change as the mission unfolds.)

6.5.5 Monitoring Behavior Run States and Messages During Mission Execution

The run states for each behavior, are wrapped up on each iteration by the helm into a single string and published in the variable `IVPHELM_SUMMARY`. This variable is subscribed for by the `uHelmScope` tool and behavior states are parsed from this variable and summarized in the main output, as in lines 12-17 in Listing **??** on page **??**. These lines are provided in the below excerpt:

```
12  Behaviors Active: ---------- (1)
13    waypt_survey (13.0) (pwt=100.00) (pcs=1227) (cpu=0.01) (upd=0/0)
14  Behaviors Running: --------- (0)
15  Behaviors Idle: ------------ (1)
16    waypt_return (22.8)
17  Behaviors Completed: ------- (0)
```

Behaviors are grouped into the four possible states, with a summary line for each state, e.g., lines 12, 14, 15, 17, containing the number of behaviors in that state in parentheses at the end of the line. Each behavior configured for the helm shows up on a dedicated line in the appropriate group, e.g., lines 13 and 16. In these lines immediately following the behavior name, the number of seconds is displayed in parentheses indicating how long the behavior has been in that state.

6.6 Behavior Reconciliation in the IvP Helm - Multi-Objective Optimization

6.6.1 IvP Functions

IvP functions are produced by behaviors to influence the decision produced by the helm on the current iteration (see Figure 12, p. 63). The decision is typically comprised of the desired `heading`, `speed`, and `depth` but the helm decision space could be comprised of any arbitrary configuration (see section 5.4, p. 50). Some points about IvP functions:

- IvP functions are piecewise linearly defined. Each piece is defined by an interval over some subset of the decision space, and there is a linear function associated with each piece (see Figure 18).
- IvP functions are an *approximation* of an underlying function. The linear function for a single piece is the best linear approximation of the underlying function for the portion of the domain covered by that piece.
- IvP domains are discrete with an upper and lower bound for each variable, so an IvP function *may* achieve zero-error in approximating an underlying function by associating a piece with each point in the domain. Behaviors seldom need to do so in practice however.
- The Ivp function construct and IvP solver are generalizable to N dimensions.
- The pieces in IvP functions need not be uniform size or shape. More pieces can be dedicated to parts of the domain that are harder to approximate with linear functions.
- IvP functions need only be defined over a subset of the domain. Behaviors are not affected if the helm is configured for additional variables that a behavior may not care about. Behaviors that produce functions solely over vehicle `depth` are perfectly ok.

How are IvP functions built? The IvP Build Toolbox is a set of tools for creating IvP functions based on any underlying function defined over an IvP Domain. Many, if not all of the behaviors in this document make use of this toolbox, and authors of new behaviors have this at their disposal. A primary component of writing a new behavior is the development of the "underlying function", the function approximated by an IvP function with the help of the toolbox. The underlying function represents the relationship between a candidate helm decision and the expected utility with respect to the behavior's objectives. The IvP Toolbox is not covered in detail in this document, but an overview is given below.

6.6.2 The IvP Build Toolbox

The IvP Toolbox is a set of tools (a C++ library) for building IvP functions. It is typically utilized by behavior authors in a sequence of library calls within a behavior's (C++) implementation. There are two sets of tools - the *Reflector* tools for building IvP functions in N dimensions, and the *ZAIC* tools for building IvP functions in one dimension as a special case. The Reflector tools work by making available a function to be approximated by an IvP function. The tools simply need this function for sampling. Consider the Gaussian function rendered below in Figure 17:

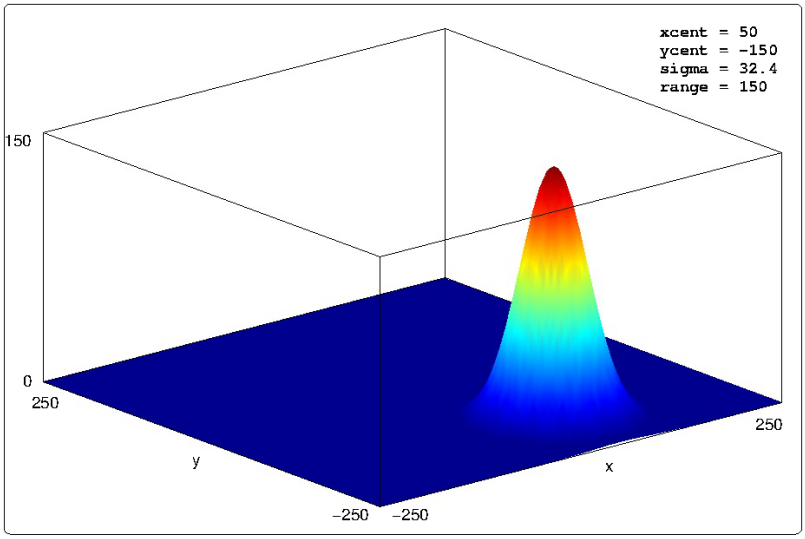
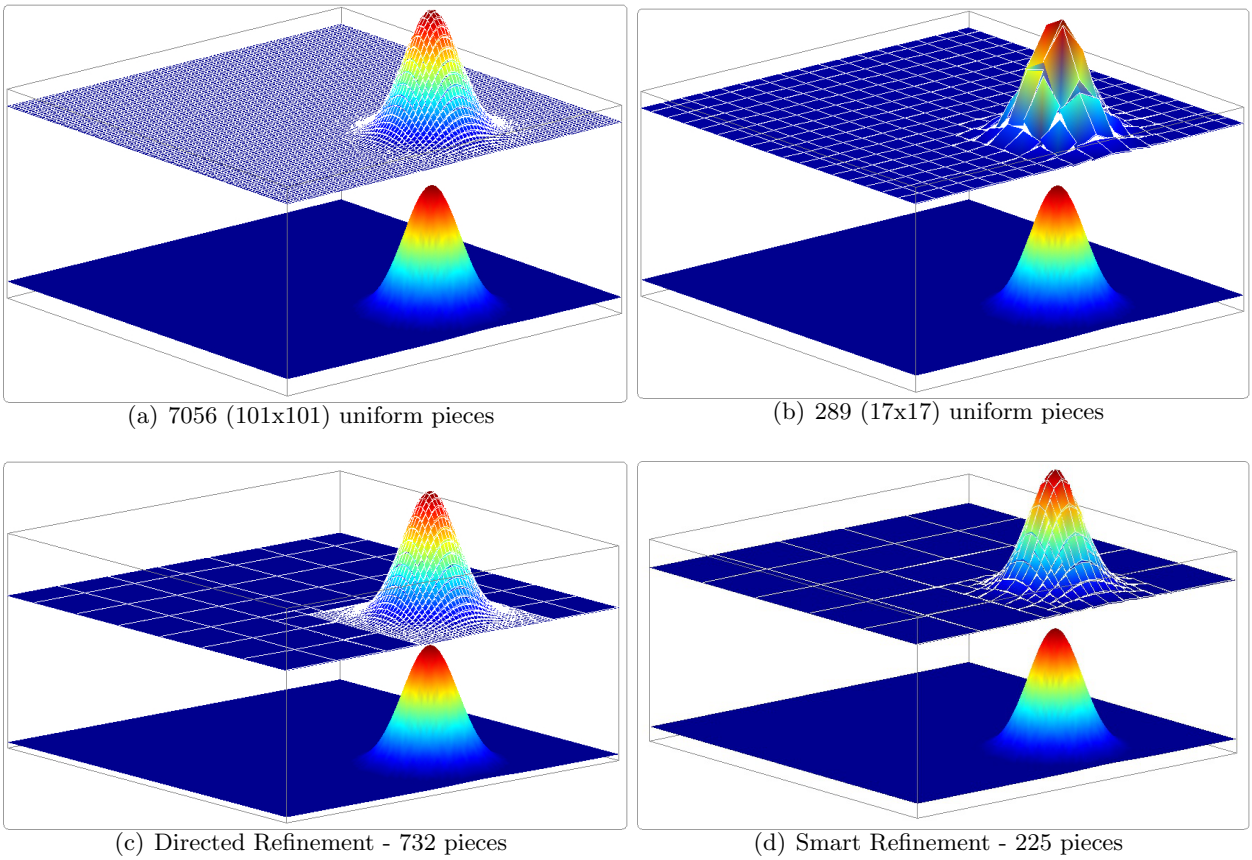


Figure 17: A rendering of the function $f(x,y) = Ae^{-(\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})}$ where $A =$ `range` $= 150$, $\sigma =$ `sigma` $= 32.4$, $x_0 =$ `xcent` $= 50$, $y_0 =$ `ycent` $= -150$. The domain here for x and y ranges from -250 to 250.

The `'x'` and `'y'` variables, each with a range of $[-250, 250]$, are discrete, taking on integer values. The domain therefore contains $501^2 = 251,001$ points, or possible decisions. The IvP Build Toolbox can generate an IvP function approximating this function over this domain by using a uniform piece size, as rendered in Figure 18(a) and 18(b). The difference in these two figures is only the size of the piece. More pieces (Figure 18(a)) results in a more accurate approximation of the underlying function, but takes longer to generate and creates further work for the IvP solver when the functions are combined. IvP functions need not use uniformly sized pieces.



(a) 7056 (101x101) uniform pieces

(b) 289 (17x17) uniform pieces

(c) Directed Refinement - 732 pieces

(d) Smart Refinement - 225 pieces

Figure 18: **A rendering of four different IvP functions approximating the same underlying function**: The function in (a) uses a uniform distribution of 7056 pieces. The function in (b) uses a uniform distribution of 1024 pieces. The function in (c) was created by first building a uniform distribution of 49 pieces and then focusing the refinement on a sub-domain of the function. This is called directed-refinement in the IvP Build toolbox. The function in (d) was created by first building a uniform function of 25 pieces and repeatedly refining the function based on which pieces were noted to have a poor fit to the underlying function. This is termed smart-refinement in the IvP Build toolbox.

By using the *directed refinement* option in the IvP Build Toolbox, an initially uniform IvP function can be further refined with more pieces over a sub-domain directed by the caller, with smaller uniform pieces of the caller's choosing. This is rendered in Figure 18(c). Using this tool requires the caller to have some idea where, in the sub-domain, further refinement is needed or desired. Often a behavior author indeed has this insight. For example, if one of the domain variables is vehicle heading, it may be good to have a fine refinement in the neighborhood of heading values close to the vehicle's current heading.

In other situations, insight into where further refinement is needed may not be available to the caller. In these cases, using the *smart refinement* option of the IvP Build Toolbox, an initially uniform IvP function may be further refined by asking the toolbox to automatically "grade" the pieces as they are being created. The grading is in terms of how accurate the linear fit is between the piece's linear function and the underlying function over the sub-domain for that piece. A priority

queue is maintained based on the grades, and pieces where poor fits are noted, are automatically refined further, up to a maximum piece limit chosen by the caller. This is rendered in Figure 18(d).

The Reflector tools work similarly in N dimensions and on multi-modal functions. The only requirement for using the Reflector tool is to provide it with access to the underlying function. Since the tool repetitively samples this function, a central challenge to the user of the toolbox is to develop a fast implementation of the function. In terms of the time consumed in generating IvP functions with the Reflector tool, the sampling of the underlying function is typically the long pole in the tent.

6.6.3 The IvP Solver and Behavior Priority Weights

The IvP Solver collects a set of weighted IvP functions produced by each of the behaviors and finds a point in the decision space that optimizes the weighted combination. If each IvP objective function is represented by $f_i(\vec{x})$, and the weight of each function is given by w_i, the solution to a problem with k functions is given by:

$$\vec{x}^* = \underset{\vec{x}}{\text{argmax}} \sum_{i=0}^{k-1} w_i f_i(\vec{x})$$

The algorithm is described in detail in [3], but is summarized in the following few points.

- *The search tree*: The structure of the search algorithm is branch-and-bound. The search tree is comprised of an IvP function at each layer, and the nodes at each layer are comprised of the individual pieces from the function at that layer. A leaf node represents a single piece from each function. A node in the tree is realizable if the piece from that node and its ancestors intersect, i.e., share common points in the decision space.

- *Global optimality*: Each point in the decision space is in exactly one piece in each IvP function and is thus in exactly one leaf node of the search tree. If the search tree is expanded fully, or pruned properly (only when the pruned out sub-tree does not contain the optimal solution), then the search is guaranteed to produce the globally optimal solution. The search algorithm employed by the IvP solver does indeed start with a fully expanded tree, and utilizes proper pruning to guarantee global optimality. The algorithm does allow for a parameter for guaranteed limited back-off from the global optimality - a quicker solution with a guarantee of being within a fixed percent of global optima. This option is not exposed to the IvP Helm which always finds the global optimum.

- *Initial solution*: A key factor of an effective branch-and-bound algorithm is seeding the search with a decent initial solution. In the IvP Helm, the initial solution used is the solution (typically `heading`, `speed`, `depth`) generated on the previous helm iteration. Upon casual observation this appears to provide a speed-up by about a factor of two.

In cases where there is a "tie" between optimal decisions, the solution generated by the solver is non-deterministic. This is mitigated somewhat by the fact that the solution is seeded with the output of the previous iteration as discussed above.

6.6.4 Monitoring the IvP Solver During Mission Execution

The performance of the solver can be monitored with the `uHelmScope` tool described in Section 11. The output shown below in Listing 13 is an excerpt of the full output shown in Listing **??** on page **??**. On line 5, the total time needed to solve the multi-objective optimization problem is given in seconds, and the max time need for all recorded loops is given in parentheses. It is zero here since there is only one objective function in this example. On line 6 is the total time for creating the IvP functions in all behaviors, with the max across all iterations in parentheses. On line 7 is the total loop time - the sum of the previous two lines. Active behaviors display useful information regarding the IvP solver. For example, on line 13, the Survey waypoint behavior had a priority weight of 100 and generated 1,227 pieces, taking 0.01 seconds of CPU time to create.

Listing 13 - Example `uHelmScope` *output containing information about the IvP solver.*

```
 1  ==============   uHelmScope Report  ============== DRIVE  (17)
 2    Helm Iteration: 66      (hz=0.38)(5)  (hz=0.35)(66)  (hz=0.56)(max)
 3    IvP functions:  1
 4    Mode(s):        Surveying
 5    SolveTime:      0.00    (max=0.00)
 6    CreateTime:     0.02    (max=0.02)
 7    LoopTime:       0.02    (max=0.02)
 8    Halted:         false   (0 warnings)
 9  Helm Decision: [speed,0,4,21] [course,0,359,360]
10    speed = 3.00
11    course = 177.00
12  Behaviors Active: ---------- (1)
13    waypt_survey (13.0) (pwt=100.00) (pcs=1227) (cpu=0.01) (upd=0/0)
14  Behaviors Running: --------- (0)
15  Behaviors Idle: ------------ (1)
16    waypt_return (22.8)
17  Behaviors Completed: ------- (0)
18
```

The solver can be additionally monitored and analyzed through the two MOOS variables `LOOP_CPU` and `CREATE_CPU` published on each helm iteration. The former indicates the system wall time for building each IvP function and solving the multi-objective optimization problem, and the latter indicates just the time to create the IvP functions.

7 Properties of Helm Behaviors

The objective of this section is to describe properties common to all IvP Helm behaviors, describe how to overload standard functions for 3rd party behaviors, and to provide a detailed simple example of a behavior. It builds on the discussion from Chapter 6. The focus in this section is an expansion of detail of Step 3 in Figure 12 on page 63.

7.1 Brief Overview

Behaviors are implemented as C++ classes with the helm having one or more instances at runtime, each with a unique descriptor. The properties and implemented functions of a particular behavior are partly derived from the `IvPBehavior` superclass, shown in Figure 19. The is-a relationship of a derived class provides a form of code re-use as well as a common interface for constructing mission files with behaviors.

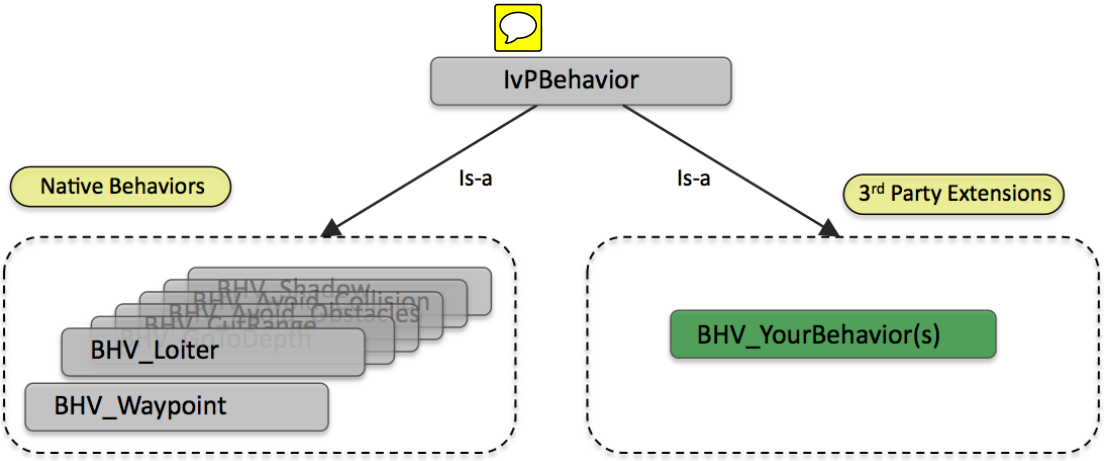


Figure 19: **Behavior inheritance**: Behaviors are derived from the `IvPBehavior` superclass. The native behaviors are the behaviors distributed with the helm. New behaviors also need to be a subclass of the IvPBehavior class to work with the helm. Certain virtual functions invoked by the helm may be optionally but typically overloaded in all new behaviors. Other private functions may be invoked within a behavior function as a way of facilitating common tasks involved in implementing a behavior.

The `IvPBehavior` class provides five virtual functions which are typically overloaded in a particular behavior implementation:

- The `setParam()` function: parameter-value pairs are handled to configure a behavior's unique properties distinct from its superclass.
- The `onRunState()` function: the meat of a behavior implementation, performed when the behavior has met its conditions for running, with the output being an objective function and a possibly empty set of variable-value pairs for posting to the `MOOSDB`.
- The `onIdleState()` function: what the behavior does when it has not met its run conditions. It may involve updating internal state history, generation of variable-value pairs for posting to the `MOOSDB`, or absolutely nothing at all.

- The `onIdleToRunState()` function: invoked once by the helm upon transitioning from the idle to running state (compared to the `onRunState()` function which is invoked on *each* helm iteration where the behavior has met its conditions).
- The `onRunToIdleState()` function: invoked once by the helm upon transitioning from the running to idle state (compared to the `onIdleState()` function which is invoked on *each* helm iteration where the behavior has not met its conditions).

This section discusses the properties of the `IvPBehavior` superclass that an author of a third-party behavior needs to be aware of in implementing new behaviors. It is also relevant material for users of the native behaviors as it details general properties.

7.2 Parameters Common to All IvP Behaviors

A behavior has a standard set of parameters defined at the `IvPBehavior` level as well as unique parameters defined at the subclass level. By configuring a behavior during mission planning, the setting of parameters is the primary venue for affecting the overall autonomy behavior in a vehicle. Parameters are set in the behavior file, but can also be dynamically altered once the mission has commenced. A parameter is set with a single line of the form:

```
parameter = value
```

The left-hand side, the parameter component, is case insensitive, while the value component is typically case sensitive. This was discussed in depth in Section 6.3. In this section, the parameters defined at the superclass level and available to all behaviors are exhaustively listed and discussed. Each behavior typically augments these parameters with new ones unique to the behavior, and in the next section the issue of implementing new parameters by overloading the `setParam()` function is addressed.

7.2.1 A Summary of the Full Set of General Behavior Parameters

The following parameters are defined for all behaviors at the superclass level. They are listed here for reference - certain related aspects are discussed in further detail in other sections.

`name`: The name of the behavior - should be unique between all behaviors. Due to the implementation of behavior templating, spawned behavior take on a new name by concatenating on the end of a base name. For this reason all configured behavior names could be regarded as base names. The check for uniqueness includes hypothetical extension. Thus the names `loiter` and `loiter_two` are not regarded as safe since the first could potentially grow into the latter with the contcatenation of `_two`. Logging and output sent to the helm console during operation will organize information by the behavior name.

`priority`: The priority weight of the produced objective function. The default value is 100. A behavior may also be implemented to determine its own priority weight depending on information about the world.

duration: The time in seconds that the behavior will remain running before declaring completion. If no duration value is provided, the behavior will never time-out. The clock starts ticking once the behavior satisfies its run conditions (becoming non-idle) the first time. *Should the behavior switch between running and idle states, the clock keeps ticking even during the idle periods.* See Section 7.2.3 for more detail.

duration_status: If the duration parameter is set, the remaining duration time, in seconds, can be posted by naming a duration_status variable. This variable will be update/posted only when the behavior is in the running state. See Section 7.2.3 for more detail.

duration_reset: This parameter takes a variable-pair such as MY_RESET=true. If the duration parameter is set, the duration clock is reset when the variable is posted to the MOOSDB with the specified value. Each time such a post is noted, the duration clock is reset. See Section 7.2.3 for more detail.

post_mapping: This parameter takes a comma-separated pair such as WPT_STAT, WAYPT_STATUS where the left-hand value is a variable normally posted by the behavior, and the right-hand value is an alternative variable name to be used. There is no error-checking to ensure that the left-hand value names a variable actually posted by the behavior. Transitive relationships are not respected. For example, if the two re-mappings are declared, FOO,BAR, and BAR,CAR, FOO will be posted as BAR, not CAR. To disable the normal posting of a variable FOO, use post_mapping psvar= FOO,SILENT.

duration_idle_decay: If this parameter is `false` the duration clock is paused when the vehicle is in the "idle" state. The default value is `true`. See Section 7.2.3 for more detail.

condition: This parameter specifies a condition that must be met for the behavior to be active. Conditions are checked for each behavior at the beginning of each control loop iteration. Conditions are based on current MOOS variables, such as `STATE = normal` or ($(K \leq 4)$). More than one condition may be provided, as a convenience, treated collectively as a single conjunctive condition. The helm automatically subscribes for any condition variables. See Section 6.5.1 for more detail on run conditions.

runflag: This parameter specifies a variable and a value to be posted when the behavior has met all its conditions for being in the *running* state. It is only posted if the behavior, on the previous helm iteration, was not in the *running* state. It is an equal-separated pair such as TRANSITING=true. More than one flag may be provided. These can be used to satisfy or block the conditions of other behaviors. See Section 6.5.4 on page 73 for more detail on posting flags to the MOOSDB from the helm.

idleflag: This parameter specifies a variable and a value to be posted when the behavior is in the *idle* state. See the Section 6.5.3 for more on run states. It is only posted if the behavior, on the previous helm iteration, was not in the *idle* state. It is an equal-separated pair such as WAITING=true. More than one flag may be provided. These can be used to satisfy or block the conditions of other behaviors. See Section 6.5.4 on page 73 for more detail on posting flags to the MOOSDB from the helm.

activeflag: This parameter specifies a variable and a value to be posted when the behavior is in the *active* state. See the Section 6.5.3 for more on run states. It is only posted if the behavior, on the previous helm iteration, was not in the *avtive* state. It is an equal-separated pair such as TRANSITING=true. More than one flag may be provided. These can be used to satisfy or block the conditions of other behaviors. See Section 6.5.4 on page 73 for more detail on posting flags to the MOOSDB from the helm.

inactiveflag: This parameter specifies a variable and a value to be posted when the behavior is *not* in the *active* state. See the Section 6.5.3 for more on run states. It is only posted if the behavior, on the previous helm iteration, was in the *active* state. It is a equal-separated pair such as OUT_OF_RANGE=true. More than one flag may be specified by the user in the behavior configuration. These can be used to satisfy or block the conditions of other behaviors. See Section 6.5.4 on page 73 for more detail on posting flags to the MOOSDB from the helm.

endflag: This parameter specifies a variable and a value to be posted when the behavior has set the `completed` state variable to be true. The circumstances causing completion are unique to the individual behavior. However, if the behavior has a duration specified, the `completed` flag is set to true when the duration is exceeded. The value of this parameter is a equal-separated pair such as ARRIVED_HOME=true. Once the completed flag is set to true for a behavior, it remains inactive thereafter, regardless of future events, barring a complete helm restart. See Section 6.5.4 on page 73 for more detail on posting flags to the MOOSDB from the helm.

updates: This parameter specifies a variable from which updates to behavior configuration parameters are read from after the behavior has been initially instantiated and configured at the helm startup time. Any parameter and value pair that would have been legal at startup time is legal at runtime. The syntax for this string is a #-separated list of parameter-value pairs: `"param=value # param=value # ... # param=value"`. This is one of the primary hooks to the helm for mission control - the other being the behavior conditions described above. See Section 7.2.2 for more detail.

nostarve: The nostarve parameter allows a behavior to assert a maximum staleness for one or more MOOS variables, i.e., the time since the variable was last updated. The syntax for this parameter is a comma-separated pair `"variable, ..., variable, value"`, where last component in the list is the time value given in seconds. See Section 7.2.5 on page 86 for more detail.

perpatual: Setting the perpetual parameter to `true` allows the behavior to continue to run even after it has completed and posted its end flags. The parameter value is not case sensitive and the only two legal values are `true` and `false`. See Section 7.2.4 for more detail.

templating: The templating parameter may be used to turn a behavior specification into a template for spawning new behaviors dynamically after the helm has been launched. Instantiation requests are received via the updates parameter described in Section 7.7.

7.2.2 Altering Behavior Parameters Dynamically with the updates Parameter

The parameters of a behavior can be made to allow dynamic modifications - after the helm has been launched and executing the initial mission in the behavior file. The modifications come in a single MOOS variable specified by the parameter updates. For example, consider the simple waypoint behavior configuration below in Listing 14. The return point is the (0,0) point in local coordinates, and return speed is 2.0 meters/second. When the conditions are met, this is what will be executed.

Listing 14 - An example behavior configuration using the updates parameter.

```
 0  Behavior = BHV_Waypoint
 1  {
 2    name      = WAYPT_RETURN
 3    priority  = 100
 4    speed     = 2.0
 5    radius    = 8.0
 6    points    = 0,0
 7    updates   = RETURN_UPDATES
 8    condition = RETURN = true
 9    condition = DEPLOY = true
10  }
```

If, during the course of events, a different return point or speed is desired, this behavior can be altered dynamically by writing to the variable specified by the updates parameter, in this case the variable RETURN_UPDATES (line 7 in Listing 14). The syntax for this variable is of the form:

```
parameter = value # parameter = value # ... # parameter = value
```

White space is ignored. The '#' character is treated as special for parsing the line into separate parameter-value pairs. It cannot be part of a parameter component or value component. For example, the return point and speed for this behavior could be altered by any other MOOS process that writes to the MOOS variable:

```
RETURN_UPDATES = "points=50,50 # speed = 1.5"
```

Each parameter-value pair is passed to the same parameter setting routines used by the behavior on initialization. The only difference is that an erroneous parameter-value pair will simply be ignored as opposed to halting the helm as done on startup. If a faulty parameter-value pair is encountered, a warning will be written to the variable BHV_WARNING. For example:

```
BHV_WARNING = "Faulty update for behavior: WAYPT_RETURN. Bad parameter(s): speed."
```

Note that a check for parameter updates is made at the outset of helm iteration loop for a behavior (Figure 19) with the call `checkUpdates()`. Any updates received by the helm on the current iteration will be applied prior to behavior execution and in effect for the current iteration.

7.2.3 Limiting Behavior Duration with the DURATION Parameter

The duration parameter specifies a time period in seconds before a behavior times out and permanently enters the completed state (Figure 20). If left unspecified, there is no time limit to the behavior. By default, the duration clock begins ticking as soon as the helm is in drive. The duration clock remains ticking when or if the behavior subsequently enters the idle state. It even remains ticking if the helm temporarily parks. When a timeout occurs, end flags are posted. The behavior can be configured to post the time remaining before a timeout with the duration_status parameter. The forms for each are:

```
duration        =  value (positive numerical)
duration_status =  value (variable name)
```

Note that the duration status variable will only be published/updated when the behavior is in the running state. The duration status is rounded to the nearest integer until less than ten seconds remain, after which the time is posted out to two decimal places. The behavior can be configured to have the duration clock pause when it is in the idle state with the following:

```
duration_idle_decay = false   // The default is true
```

Configured in the above manner, a behavior's duration clock will remain paused until it's conditions are met. The behavior may also be configured to allow for the duration clock to be reset upon the writing of a MOOS variable with a particular value. For example:

```
duration_reset = BRAVO_TIMER_RESET=true
```

The behavior checks for and notes that the variable-value pair holds true and the duration clock is then reset to the original duration value. The behavior also marks the time at which the variable-value pair was noted to have held true. Thus there is no need to "un-set" the variable-value pair, e.g., setting BRAVO_TIMER_RESET=false, to allow the duration clock to resume its count-down.

7.2.4 The perpetual Parameter

When a behavior enters the *completed* state, it by default remains in that state with no chance to change. When the perpetual parameter is set to `true`, a behavior that is declared to be complete does not actually enter the complete state but performs all the other activity normally associated with completion, such as the posting of end flags. See Section 6.5.4 for more detail on posting flags to the MOOSDB from the helm. The default value for perpetual is `false`. The form for this parameter is:

```
perpetual = value
```

The value component is case insensitive, and the only legal values are either `true` or `false`. A behavior using the `duration` parameter with `perpetual` set to `true` will post its end flags upon time out, but will reset its clock and begin the count-down once more the next time its run conditions are met, i.e., enters the running state. Typically when a behavior is used in this way, it also posts an endflag that would put itself in the idle state, waiting for an external event.

7.2.5 Detection of Stale Variables with the `nostarve` Parameter

A behavior utilizing a variable generated by a MOOS process outside the helm, may require the variable to be sufficiently up-to-date. The staleness of a variable is the time since it was last written to by any process. The `nostarve` parameter allows the mission writer to set a staleness threshold. The form for this parameters is:

```
nostarve = variable_1, ..., variable_n, duration
```

The value of this parameter is a comma-separated list such as `nostarve = NAV_X, NAV_Y, 5.0`. The variable components name MOOS variables and the duration component, the last entry in the list, represents the tolerated staleness in seconds. If staleness is detected, a behavior failure condition is triggered which will trigger the helm to post all-stop values and relinquish to manual control.

7.3 Overloading the `setParam()` Function in New Behaviors

The `setParam()` function is a virtual function defined in the `IvPBehavior` class, with parameters implemented in the superclass (Section 7.2) handled in the superclass version of this function:

```
bool IvPBehavior::setParam(string parameter, string value);
```

The `setParam()` function should return `true` if the parameter is recognized and the value is in an acceptable form. In the rare case that a new behavior has no additional parameters, leaving this function undefined in the subclass is appropriate. The example below in Listing 15 gives an example for a fictional behavior `BHV_YourBehavior` having a single parameter `period`.

Listing 15 - An example `setParam()` *implementation for fictional* `BHV_YourBehavior`.

```
 0  bool BHV_YourBehavior::setParam(string param, string value)
 1  {
 2    if(param == "period") {
 3      double time_value = atof(value.c_str());
 4      if((time_value < 0) || (!isNumber(value)))
 5        return(false);
 6      m_period = time_value;
 7      return(true);
 8    }
 9    return(false);
10  }
```

Since the `period` parameter refers to a time period, a check is made on line 4 that the value component indeed is a positive number. (The `atof()` function on line 6, which converts an ASCII string to a floating point value, returns zero when passed a non-numerical string, therefore the `isNumber()` function is also used to ensure the string represented by `value` represents a numerical value.) A behavior implementation of this function without sufficient syntax or semantic checking simply runs the risk that faulty parameters are not detected at the time of helm launch, or during dynamic updates. Solid checking in this function will reduce debugging headaches down the road.

7.4 Behavior Functions Invoked by the Helm

The `IvPBehavior` superclass implements a number of functions invoked by the helm on each iteration. Two of these functions are overloadable as described previously - the `onRunState()` and `onIdleState()` functions. The basic flow of calls to a behavior from the helm are shown in Figure 20. These are discussed in more detail later in the section, but the idea is to execute certain behavior functions based on the *activity state*, which may be one of the four states depicted.

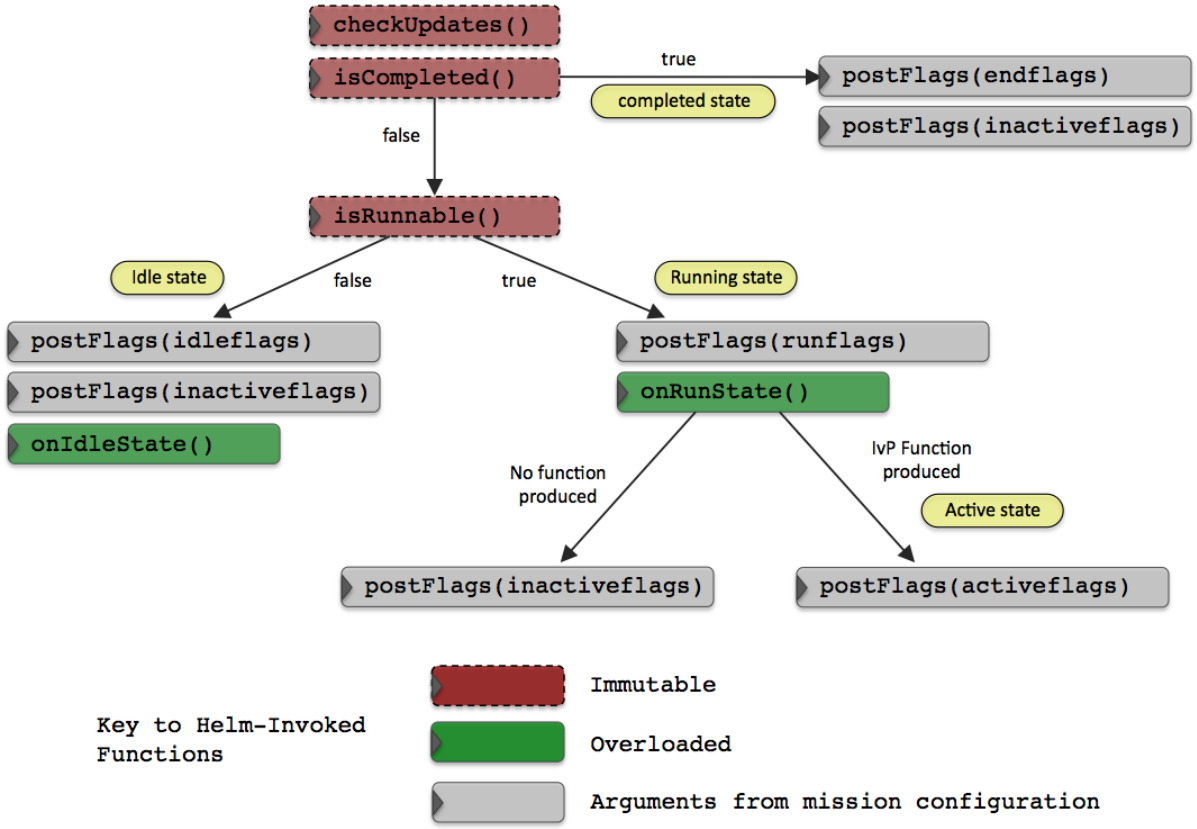


Figure 20: **Behavior function-calls by the helm**: The helm invokes a sequence of functions on each behavior on each iteration of the helm. The sequence of calls is dependent on what the behavior returns, and reflects the behaviors activity state. Certain functions are immutable and can not be overloaded by a behavior author. Two key functions, `onRunState()` and `onIdleState()` can be indeed overloaded as the usual hook for an author to provide the implementation of a behavior. The `postFlags()` function is also immutable, but the parameters (flags) are provided in the helm configuration (`*.bhv`) file.

An *idle* behavior is one that has not met its conditions for running. A *completed* behavior is one that has reached its objectives or exceeded its duration. A *running* behavior is one that has not yet completed, has met its run conditions, but may still opt not to produce any output. An *active* behavior is one that is running and is producing output in the form of an objective function.

The types of functions defined at the superclass level fall into one of the three categories below, only the first two of which are shown in Figure 20:

- Helm-invoked immutable functions - functions invoked by the helm on each iteration that the author of a new behavior may not re-implement.
- Helm-invoked overloadable functions - functions invoked by the helm that an author of a new behavior typically re-implements of overloads.
- User-invoked functions - functions invoked within a behavior implementation.

The user-invoked functions are utilities for common operations typically invoked within the

implementation of the `onRunState()` and `onIdleState()` functions written by the behavior author.

7.4.1 Helm-Invoked Immutable Functions

These functions, implemented in the `IvPBehavior` superclass, are called by the helm but are *not* defined as virtual functions which means that attempts to overload them in a new behavior implementation will be ignored. See Figure 20 regarding the sequence of these function calls.

`void checkUpdates()`: This function is called first on each iteration to handle requested dynamic changes in the behavior configuration. This needs to be the very first function applied to a behavior on the helm iteration so any requested changes to the behavior parameters may be applied on the present iteration. See Section 7.2.2 for more on dynamic behavior configuration with the `updates` parameter.

`bool isComplete()`: This function simply returns a Boolean indicating whether the behavior was put into the *complete* state during a prior iteration.

`bool isRunnable()`: Determines if a behavior is in the *running* state or not. Within this function call four things are checked: (a) if the `duration` is set, the duration time remaining is checked for timeout, (b) variables that are monitored for staleness are checked against (Section 7.2.5). (c) the run conditions must be met. (d) the behavior's decision domain (IvP domain) is a proper subset of the helm's configured IvP domain. See Section 6.5.1 for more detail on run conditions.

`void postFlags(string flag_type)`: This function will post flags depending on whether the value of `flag_type` is set to `"idleflags"`, `"runflags"`, `"activeflags"`, `"inactiveflags"`, or `"endflags"`. Although this function is immutable, not overloadable by subclass implementations, its effect is indeed mutable since the flags are specified in the mission configuration `*.bhv` file. See Section 6.5.4 for more detail on posting flags to the `MOOSDB` from the helm.

7.4.2 Helm-Invoked Overloaded Functions

These are functions called by the helm. They are defined as virtual functions so that a behavior author may overload them. Typically the bulk of writing a new behavior resides in implementing these three functions.

`IvPFunction* onRunState()`: The `onRunState()` function is called by the helm when deemed to be in the *running* state (Figure 20). The bulk of the work in implementing a new behavior is in this function implementation, and is the subject of Section 7.6.

`void onIdleState()`: This function is called by the helm when deemed to be in the *idle* state (Figure 20). Many behaviors are implemented with this function left undefined, but it is a useful hook to have in many cases.

`bool setParam(string, string)`: This function is called by the helm when the behavior is first instantiated with the set of parameter and parameter values provided in the behavior file. It is also called by the helm within the `checkUpdates()` function to apply parameter updates dynamically.

7.5 Local Behavior Utility Functions

The bulk of the work done in implementing a new behavior is in the implementation of the `onIdleState()` and `onRunState()` functions. The utility functions described below are designed to aid in that implementation and are generally "protected" functions, that is callable only from within the code of another function in the behavior, such as the `onRunState()` and `onIdleState()` functions, and not invoked by the helm.

7.5.1 Summary of Implementor-Invoked Utility Functions

The following is summary of utility functions implemented at the `IvPBehavior` superclass level.

`void setComplete()`: The notion of what it means for a behavior to be "complete" is largely an issue specific to an individual behavior. When or if this state is reached, a call to `setComplete()` can be made and end flags will be posted, and the behavior will be permanently put into the *completed* state unless the `perpetual` parameter is set to `true`.

`void addInfoVars(string var_names)`: The helm will register for variables from the `MOOSDB` on a need-only basis, and a behavior is obligated to inform the helm that certain variables are needed on its behalf. A call to the `addInfoVars()` function can be made from anywhere with a behavior implementation to declare needed variables. This can be one call per variable, or the string argument can be a comma-separated list of variables. The most common point of invoking this function is within a behavior's constructor since needed variables are typically known at the point of instantiation. More on this issue in Section 7.5.3.

`double getBufferDoubleVal(string varname, bool& result)`: Query the `info_buffer` for the latest (double) value for a given variable named by the string argument. The bool argument indicates whether the queried variable was found in the buffer. More on this in Section 7.5.2.

`double getBufferStringVal(string varname, bool& result)`: Query the `info_buffer` for the latest (string) value for a given variable named by the string argument. The bool argument indicates whether the queried variable was found in the buffer. More on this in Section 7.5.2.

`double getBufferCurrTime()`: Query the `info_buffer` for the current buffer local time, equivalent to the duration in seconds since the helm was launched. More on this in Section 7.5.2.

`vector<double> getBufferDoubleVector(string var, bool& result)`: Query the `info_buffer` for all changes to the variable (of type double) named by the string argument, since the last iteration. The bool argument indicates whether the queried variable was found in the buffer. More on this in Section 7.5.2.

`vector<string> getBufferStringVector(string var, bool& result)`: Query the `info_buffer` for all changes to the variable (of type string) named by the string argument, since the last iteration. The bool argument indicates whether the queried variable was found in the buffer. More on this in Section 7.5.2.

`void postMessage(string varname, string value, string key)`: The helm can post messages (variable-value pairs) to the `MOOSDB` at the end of the helm iteration. Behaviors can request such postings via a call to the `postMessage()` function where the first argument is the variable name, and the second is the variable value. The optional *key* parameter is used in conjunction with the duplication filter and by default is the empty string. See Section 5.8 for more on the duplication filter.

`void postMessage(string varname, double value, string key)`: Same as above except used when the posted variable is of type `double` rather than `string`. The optional *key* parameter is used in conjunction with the duplication filter and by default is the empty string. See Section 5.8 for more on the duplication filter.

`void postBoolMessage(string varname, bool value, string key)`: Same as above, except used when the posted variable is a `bool` rather than `string`. The optional *key* parameter is used in conjunction with the duplication filter and by default is the empty string. See Section 5.8 for more on the duplication filter.

`void postIntMessage(string varname, double value, string key)`: Same as `postMessage(string, double)` above except the numerical output is rounded to the nearest integer. This, combined with the helm's use of the *duplication filter*, can reduce the number of posts to the `MOOSDB`. The optional *key* parameter is used in conjunction with the duplication filter and by default is the empty string. See Section 5.8 for more on the duplication filter.

`void postWMessage(string warning_msg)`: Identical to the `postMessage()` function except the variable name is automatically set to `BHV_WARNING`. Provided as a matter of convenience to the caller and for uniformity in monitoring warnings.

`void postEMessage(string error_msg)`: Similar to the `postWMessage()` function except the variable name is `BHV_ERROR`. This call is for more serious problems noted by the behavior. It also results in an internal `state_ok` bit being flipped which results in the helm posting all-stop values to the actuators.

7.5.2 The Information Buffer

Behaviors do not have direct access to the `MOOSDB` - they don't read mail, and they don't post changes directly, but rather through the helm as an intermediary. The information buffer, or `info_buffer`, is a data structure maintained by the helm to reflect a subset of the information in the `MOOSDB` and made available to each behavior. This topic is hidden from a user configuring existing behaviors and can be safely skipped, but is an important issue for a behavior author implementing a new behavior. The `info_buffer` is a data structure shared by all behaviors, each behavior having an pointer to a single instance of the `InfoBuffer` class. This data structure is maintained by the helm, primarily by reading mail from the `MOOSDB` and reflecting the change onto the buffer on each helm iteration, before the helm requests input from each behavior. Each behavior therefore has the exact same snapshot of a subset of the `MOOSDB`. A behavior author needs to know two things - how to ensure that certain variables show up in the buffer, and how to access that information from within the behavior. These two issues are discussed next.

7.5.3 Requesting the Inclusion of a Variable in the Information Buffer

A variable can be specifically requested for inclusion in the `info_buffer` by invoking the following function:

```
void  IvPBehavior::addInfoVars(string varnames)
```

The string argument is either a single MOOS variable or a comma-separated list of variables. Duplicate requests are simply ignored. Typically such calls are invoked in a behavior's constructor, but may be done dynamically at any point after the helm is running. The helm will simply register with the `MOOSDB` for the requested variable at the end of the current iteration. Certain variables are registered for automatically on behalf of the behavior. All variables referenced in run conditions will be registered and accessible in the buffer. Variables named in the `updates` and `nostarve` parameters will also be automatically registered.

7.5.4 Accessing Variable Information from the Information Buffer

A variable value can be queried from the buffer with one of the following two function calls, depending on whether the variable is of type double or string.

```
string  IvPBehavior::getBufferStringVal(string varname, bool& result)
double  IvPBehavior::getBufferDoubleVal(string varname, bool& result)
```

The first string argument is the variable name, and the second argument is a reference to a Boolean variable which, upon the function return, will indicate whether the queried variable was found in the buffer. A duration value indicating the elapsed time since the variable was last changed in the buffer can be obtained from the following function call:

```
double  IvPBehavior::getBufferTimeVal(string varname);
```

The string argument is the variable name. The returned value should be exactly zero if this variable was updated by new mail received by the helm at the beginning of the current iteration. If the variable name is not found in the buffer, the return value is -1. The "current" buffer time, equivalent to the cumulative time in seconds since the helm was launched, can be retrieved with the following function call:

```
double  IvPBehavior::getBufferCurrTime()
```

The buffer time is a local variable of the `info_buffer` data structure. It is updated once at the beginning of the helm `OnNewMail()` loop prior to processing all new updates to the buffer from the MOOS mail stack, or at the beginning of the `Iterate()` loop if no mail is processed on the current iteration. Thus the time-stamp returned by the above call should be exactly the same for successive calls by all behaviors within a helm iteration.

The values returned by `getBufferStringVal()` and `getBufferDoubleVal()` represent the latest value of the variable in the `MOOSDB` at the point in time when the helm began its iteration and processed its mail stack. The value may have changed several times in the `MOOSDB` between iterations, and this information may be of use to a behavior. This is particularly true when a variable is being posted in pieces, or a sequence of delta changes to a data structure. In any event, this information can be recovered with the following two function calls:

```
vector<string>  IvPBehavior::getBufferStringVector(string varname, bool& result)
vector<double>  IvPBehavior::getBufferDoubleVector(string varname, bool& result)
```

They return all values updated to the buffer for a given variable since the last iteration in a vector of strings or doubles respectively. The latest change is located at the highest index of the vector. An empty vector is returned if no changes were received at the outset of the current iteration.

7.6 Overloading the `onRunState()` and `onIdleState()` Functions

The `onRunState()` function is declared as a virtual function in the `IvPBehavior` superclass intended to be overloaded by the behavior author to accomplish the primary work of the behavior. The primary behavior output is the objective function. This is what drives the vehicle. The objective function is an instance of the class `IvPFunction`, and a behavior generates an instance and returns a pointer to the object in the following function:

```
IvPFunction* onRunState()
```

This function is called automatically by the helm on the current iteration if the behavior is deemed to be in the *running* state, as depicted in Figure 20 on page 88. The invocation of `onRunState()` does not necessarily mean an objective function is returned. The behavior may opt not to for whatever reason, in which case it returns a null pointer. However, if it does generate a function, the behavior is said to be in the *active* state. The steps comprising the typical implementation of the `onRunState()` implementation can be summarized as follows:

- Get information from the `info_buffer`, and update any internal behavior state.
- Generate any messages to be posted to the `MOOSDB`.
- Produce an objective function if warranted.
- Return.

The same steps hold for the `onIdleState()` function except for producing an objective function. The first two steps have been discussed in detail. Accessing the `info_buffer` was described in Sections 7.5.2 - 7.5.4. The functions for posting messages to the `MOOSDB` from within a behavior were discussed in Section 7.5.1. Further issues regarding the posting of messages were covered in Section 6.5.4 The remaining issue to discuss is how objective functions are generated. This is covered in the IvPBuild Toolbox in a separate document.

7.7 Dynamic Behavior Spawning

In certain scenarios it may not be practical or possible to know in advance all the behaviors needed to accomplish mission objectives. For example, if the helm uses a certain kind of behavior to deal with another vehicle in its operation area, for collision avoidance or trailing etc., the identities or number of such vehicles may not be known when the mission planner is configuring the helm's behavior file. One way to circumvent this problem is to design a collision avoidance behavior, for example, to handle all known contacts. However, this has a couple drawbacks. It would entail a degree of multi-objective optimization be implemented within the behavior to produce an objective function that was comprehensive across all contacts. This would likely be much more computationally expensive

than simply generating an objective function for each contact. It also may be advantageous to have different types of collision avoidance behaviors for different contact types or collision avoidance protocols. In any event, the helm support for dynamic behavior spawning gives behavior architects and mission planners another potentially powerful option for implementing an autonomy system.

7.7.1 Behavior Specifications Viewed as Templates

The `templating` parameter may be used to turn an otherwise static behavior specification into a template for spawning new behaviors dynamically after the helm has been launched. Instantiation requests are received via the `updates` parameter described in Section 7.2.2. Updates received through this variable are normally used to change behavior parameters dynamically, but they can be further used to request the spawning of a new behavior by including the following component:

```
name = <new-behavior-name>
```

If the `<new-behavior-name>` is not the name of the behavior given in the behavior specification, and if it is not the name of a behavior already presently instantiated by the helm, the helm interprets this as a request to spawn a new behavior, if templating is enabled. Templating is enabled by including the following component in the behavior specification:

```
templating = <templating-mode>
```

The `<templating-mode>` may be set to either `"disallowed"` (the default), `"clone"`, or `"spawn"`. In the `"clone"` mode, the helm will instantiate a behavior immediately upon helm startup. In the `"spawn"` mode, the helm will not instantiate a behavior until it receives a request to do so via the `updates` parameter as described above. An example of a behavior configured to allow dynamic spawning is given in Listing 26 on page 181, taken from the Berta example mission.

For a behavior configured with templating enabled in the `"spawn"` mode, the helm will *not* spawn a behavior at the helm startup time. However, internally it will indeed spawn such a behavior, check that it can be found and built as configured, and then destroy it immediately. This means that the behavior configuration found in the `.bhv` configuration file must not have an invalid configuration. It is preferable to know at helm launch time that a behavior is misconfigured, rather than waiting for the spawning event to occur perhaps hours into a mission and being surprised that a critical behavior, such as collision avoidance, failed to be spawned.

7.7.2 Behavior Completion and Removal from the Helm

All behaviors, whether statically spawned upon helm startup, or dynamically spawned during the mission, are capable of dying and being removed from the helm. Death and removal are part of the consequences of a behavior entering the `completed` state. Behavior run states were discussed in Section 6.5.3 on page 73. A completed behavior configured with `perpetual=true` will not die upon completion. Once a behavior dies, its name is removed from the helm's internal registry of currently-spawned behaviors and a new behavior by the same name may be spawned at a future time.

7.7.3 Example Missions with Dynamic Behavior Spawning

Two example missions are provided that demonstrate the workings of dynamic behavior spawning, The Echo mission in Section 10.4, and the Berta mission in Section 10.6. The Echo mission involves a single vehicle with its helm configured to spawn dynamic behaviors of the type `BHV_BearingLine`. These behaviors do nothing more than post a viewable line segment to the `MOOSDB` between ownship and a point in the operation area. The interesting thing about this example is that the mission is configured with an event script (via `uTimerScript`) to automatically cue the spawning of 5000 behaviors over about one hour. Each behavior has a random duration of less than a minute, so behaviors are spawning and dying quite rapidly with visual confirmation via the viewable line segments.

The second example mission, the Berta mission, involves two vehicles that are loitering near one another. Periodically their loiter assignments are randomly altered (again through `uTimerScript`). The change in loiter locations repeatedly puts them on an unpredictable and random near-collision course and each vehicle needs to spawn a collision avoidance behavior. The interesting thing about this scenario is that the behavior, the `BHV_AvoidCollision` behavior, is an actual behavior of common use, unlike the `BHV_BearingLine` behavior used in the Echo mission. This example also uses the `pBasicContactMgr` to coordinate the receiving of contact reports with helm behavior spawning.

7.7.4 Examining the Helm's Life Event History

Behavior spawning, and behavior completion and removal from the helm, are two types of *life events* the helm takes note of and posts in the MOOS variable `IVPHELM_LIFE_EVENT`. A third type of life event occurs when a behavior spawning is aborted due to either a syntax error or a name collision. Monitoring life events at run time is possible by scoping on the variable `IVPHELM_LIFE_EVENT` with either `uXMS` or `uMS`. A better method is available via the `uHelmScope` application (versions 4.1 or later). It automatically registers for the `IVPHELM_LIFE_EVENT` variable and will generate a formatted report like that shown in Listing 16. In the post-mission analysis phase, the `aloghelm` application may be used to examine the life event history and will generate the same formatted report from a given alog file.

Listing 16 - A Life Event History generated with either the uHelmScope or aloghelm utilities.

```
 0        ************************************************
 1        *        Summary of Behavior Life Events         *
 2        ************************************************
 3
 4  Time    Iter  Event  Behavior      Behavior Type       Spawning Seed
 5  ------  ----  -----  ------------  ------------------  ----------------------------
 6   47.84     1  spawn  loiter        BHV_Loiter          helm startup
 7   47.84     1  spawn  waypt_return  BHV_Waypoint        helm startup
 8   47.84     1  spawn  station-keep  BHV_StationKeep     helm startup
 9  101.79   218  spawn  avd_henry     BHV_AvoidCollision  name=avd_henry#contact=henry
10  161.20   423  death  avd_henry     BHV_AvoidCollision
11  297.07   969  spawn  avd_henry     BHV_AvoidCollision  name=avd_henry#contact=henry
12  351.80  1159  death  avd_henry     BHV_AvoidCollision
13  461.37  1599  spawn  avd_henry     BHV_AvoidCollision  name=avd_henry#contact=henry
14  516.51  1795  death  avd_henry     BHV_AvoidCollision
15  644.94  2311  spawn  avd_henry     BHV_AvoidCollision  name=avd_henry#contact=henry
```

```
16  704.31  2517  death  avd_henry     BHV_AvoidCollision
17  730.02  2620  abort                BHV_AvoidCollision  name=avd_henry#foo=bar
18  825.17  3002  spawn  avd_henry     BHV_AvoidCollision  name=avd_henry#contact=henry
```

The life event history shown in Listing 16 was taken from the Berta example mission, the "gilda" vehicle, described in Section 10.6. The time-stamp reported in column one is the elapsed time between the event and the time of the helm's startup. The first three events, in lines 6-8, reflect the three static behaviors spawned when the helm was launched, in first iteration of the helm. The collision avoidance behavior was spawned each time (lines 9, 11, 13 etc.) the vehicle "henry" came within sufficiently close range. Each time the "henry" vehicle passed and opened range to a sufficient amount, the collision avoidance behavior completed and died (lines 10, 12, 14, etc.). Line 17 shows an example of an aborted spawning. This was brought about purposely by poking the MOOSDB with the Spawning Seed shown for that line. Since the collision avoidance parameter does not have a parameter "foo", the spawning failed.

Accessing the life event history via uHelmScope may be done by launching the scope with the vehicle's mission file, and hitting the 'L' key to toggle into the life event history mode. Or one may launch the scope directly into this mode via:

```
$ uHelmScope --life targ_gilda.moos
```

The same summary may also be accessed after mission completion via the log files:

```
$ aloghelm --life gilda.alog
```

Note that perhaps not all life events will be displayed when using uHelmScope, depending on when it is launched relative to pHelmIvP. When uHelmScope connects to the MOOSDB it will only receive the latest and all following posts to the variable IVPHELM_LIFE_EVENT. If uHelmScope connects after pHelmIvP is launched and put into drive, it may have missed older postings. The initial spawning events do not occur in the helm until the helm enters the DRIVE state. (See Section **??** about helm status). In the example in Listing 16, the helm apparently was in the PARK state for about 48 seconds before it was put into drive and began to execute its first iteration. The full event history should always be accessible via the log file however.

8 Behaviors of the IvP Helm

The following is a description of some single-vehicle behaviors currently written for the IvP Helm. The below is a guide for users of these behaviors. The topic of developing new behaviors is addressed in [6]. The setting of behavior parameters is the primary method for affecting the overall autonomy behavior in a vehicle. Parameters may also be dynamically altered once the mission has commenced. A parameter is set with a single line of the form:

```
parameter = value
```

The left-hand side, the parameter component, is case insensitive, while the value component is typically case sensitive. When the helm is launched, each behavior is created and the parameters are set. If a parameter setting in the behavior file references an unknown parameter, or if the value component fails a syntactic or semantic test, the line is noted and the helm ceases to launch.

Overview of Parameters Common to All Behaviors

The parameters below are common to all IvP behaviors, although they may have more relevance to some behaviors than others. They are defined in the IvPBehavior superclass of which all the behaviors described in this section are a subclass. More information on the functionality behind these parameters was given in Section 7.2.1.

Parameter	Description
NAME	A unique identifier for the behavior instance.
PRIORITY	Priority weight of the IvP function produced by behavior.
DURATION	Behavior duration, in seconds. Default is "unlimited".
DURATION_IDLE_DECAY	If true, clock will progress even if in the idle state.
DURATION_RESET	If limited duration, duration is reset when this variable is received.
DURATION_STATUS	MOOS variable informing of remaining duration, if duration is set.
POST_MAPPING	A mapping to change the default MOOS variable output of a behavior.
CONDITION	A logical condition that must satisfied for the behavior to run.
RUNFLAG	A flag (MOOSVar,Data) pair, posted when in the running state.
IDLEFLAG	A flag (MOOSVar,Data) pair, posted when in the idle state.
ACTIVEFLAG	A flag (MOOSVar,Data) pair, posted when in the active state.
INACTIVEFLAG	A flag (MOOSVar,Data) pair, posted when in the inactive state.
ENDFLAG	A flag (MOOSVar,Data) pair, posted when the behavior completes.
UPDATES	A MOOSVar from which behavior parameter updates are received.
NOSTARVE	If a given MOOSVar is stale by a given amount, an error is posted.
PERPETUAL	If true, a behavior may be reset after completion or duration timeout.
TEMPLATING	Enable the behavior spec as a template for dynamic spawning.

Table 4: Configuration parameters common to all IvP Behaviors.

Configuring One Variable Objective Functions

Several behaviors use a common tool for constructing objective functions over a single decision variable. These behaviors have a similar interface for configuring this tool, and it is described

here to avoid redundancy. Examples of behaviors that use this tool are the Waypoint, Loiter, PeriodicSurface, ConstantDepth, ConstantSpeed, ConstantHeading, and Shadow behaviors. This tool is called the *ZAIC_PEAK* tool, and is describe in more detail in [6]. This tool is designed with the objective function form shown in Figure 21 in mind, where there is an identifiable preferred single decision choice (the `summit`) with maximum utility, and then a gradual drop in utility as the variable varies from the preferred choice.

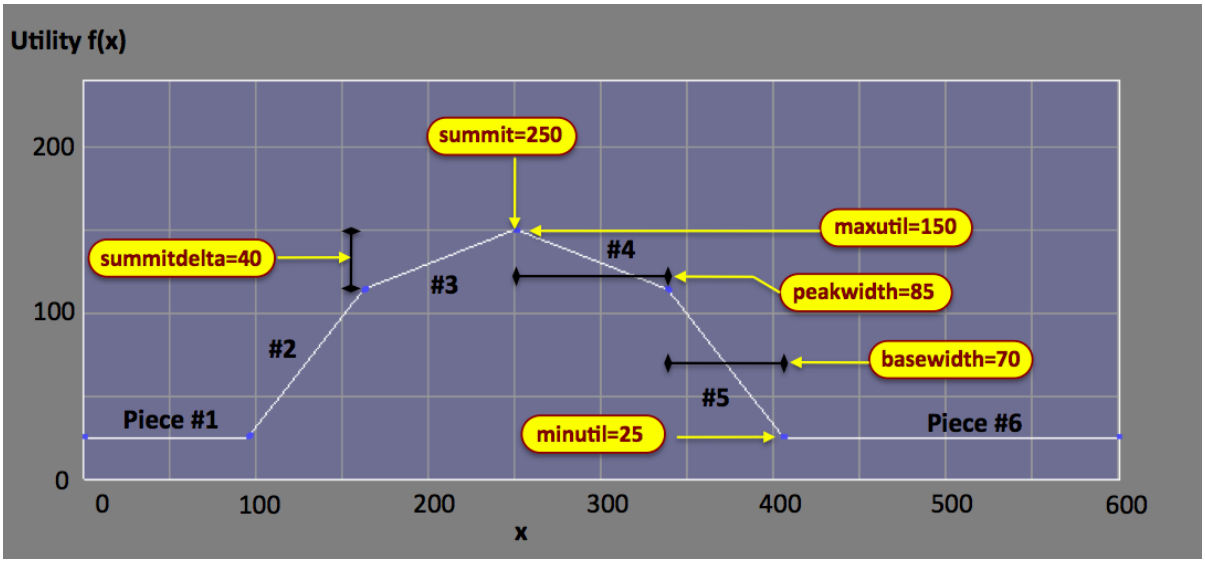


Figure 21: **The `ZAIC_PEAK` tool**: defines an IvP function over one variable defined by the four parameters shown here. In the case rendered here, the tool would create an IvP function with six pieces. The function rendered was created with `summit=180`, `peakwidth=85`, `basewidth=70`, `summitdelta=40`.

The form in which the utility drops is dependent on the settings of other parameters shown in the figure. The `summit`, `peakwidth`, and `basewidth` values are given in units native to the decision variable, while the `summitdelta`, `minutil`, and `maxutil` values are given in terms of units of utility. The latter two variables default to 0 and 100 respectively and are not typically exposed as configuration parameters in behaviors that use this tool, unlike the other four parameters.

8.1 BHV_Waypoint

The BHV_Waypoint behavior is used for transiting to a set of specified waypoint in the x-y plane. The primary parameter is the set of waypoints. Other key parameters are the inner and outer radius around each waypoint that determine what it means to have met the conditions for moving on to the next waypoint. The basic idea is shown in Figure 22.

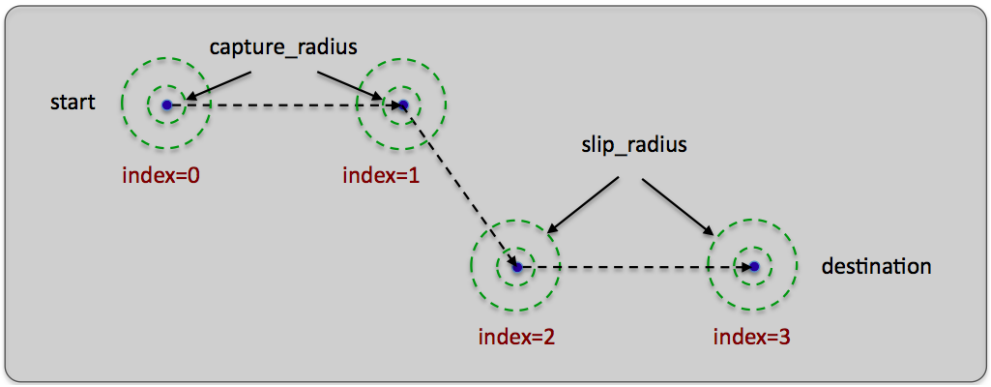


Figure 22: **The BHV_Waypoint behavior:** The waypoint behavior basic purpose is to traverse a set of waypoints. A capture radius is specified to define what it means to have achieved a waypoint, and a non-monotonic radius is specified to define what it means to be "close enough" should progress toward the waypoint be noted to degrade.

The behavior may also be configured to perform a degree of track-line following, that is, steering the vehicle not necessarily toward the next waypoint, but to a point on the line between the previous and next waypoint. This is to ensure the vehicle stays closer to this line in the face of external forces such as wind or current. The behavior may also be set to "repeat" the set of waypoints indefinitely, or a fixed number of times. The waypoints may be specified either directly at start-up, or supplied dynamically during operation of the vehicle. There are also a number of accepted geometry patterns that may be given in lieu of specific waypoints, such as polygons, lawnmower pattern and so on.

8.1.1 Brief Overview of Configuration Parameters and Variables Published

The configuration parameters and variables published collectively define the interface for the behavior. A more detailed description of usage is provided other parts of this section, and in Table 31 on page 265.

Configuration Parameters for the BHV_Waypoint Behavior

The following are the parameters for this behavior, in addition to the configuration parameters defined for all behaviors, described in Section 7.2 beginning on page 81.

Parameter	Description BHV_Waypoint
CAPTURE_RADIUS	The radius tolerance, in meters, for satisfying the arrival at a waypoint.
CYCLEFLAG	MOOS variable-value pairs posted at end of each cycle through waypoints.
LEAD	If this parameter is set, track-line following between waypoints is enabled.
LEAD_DAMPER	Distance from trackline within which the `lead` distance is stretched out.
LEAD_ON_START	The track-line following lead value in traversing to the first waypoint.
NM_RADIUS	A depricated alias for SLIP_RADIUS.
ORDER	The order in which waypoints are traversed - `"normal"`, or `"reverse"`.
POINTS	A colon separated list of x,y pairs given as points in 2D space, in meters.
POINT:	A single x,y pair given as a point in 2D space, in meters.
POLYGON	An alias for POINTS. Need not be a convex polygon.
POST_SUFFIX	A suffix tagged onto the WPT_STATUS, WPT_INDEX and CYCLE_INDEX variables.
RADIUS	An alias for CAPTURE_RADIUS.
SLIP_RADIUS	An "outer" capture radius. Arrival declared when the vehicle is in this range and the distance to the next waypoint begins to increase.
SPEED	The desired speed (m/s) at which the vehicle travels through the points.
REPEAT	The number of *extra* times traversed through the waypoints. Or `"forever"`.
VISUAL_HINTS	Hints for visual properties in variables posted intended for rendering.

Table 5: Configuration parameters for the BHV_Waypoint behavior.

Variables Published by the BHV_Waypoint Behavior

The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be supressed or changed to a different variable name using the `post_mapping` configuration parameter described in Section 7.2.1 on page 82.

MOOS Variable	Description
WPT_STAT	A comma-separated string showing the status in hitting the list of points.
WPT_INDEX	The index of the current waypoint. First point has index 0.
CYCLE_INDEX	The number of times the full set of points has been traversed, if repeating.
VIEW_POINT	A visual cue for indicating the waypoint currently heading toward.
VIEW_POINT	A visual cue for indicating the steering point, if the `lead` parameter is used.
VIEW_SEGLIST	A visual cue for rendering the full set of waypoints.

Table 6: Variables posted by the BHV_Waypoint behavior.

The following are some examples:

```
    WPT_STAT = vname=alpha,behavior-name=waypt_survey,index=1,hits=1/1,cycles=0,dist=30,eta=15
   WPT_INDEX = 3
 CYCLE_INDEX = 1
  VIEW_POINT = active,true:label,alpha's track-point:label_color,0,0.5,0:
               type,track_point:source,alphawaypt_survey:vertex_color,1,0,0:60,-152.57,0
VIEW_SEGLIST = label,alpha_waypt_survey:vertex_color,1,1,0:edge_size,1.0:
               vertex_size,2.0:60,-40:60,-160:150,-160:180,-100:150,-40
```

8.1.2 Specifying Waypoints - the `points`, `order`, and `repeat` Parameters

The waypoints may be specified explicitly as a colon-separated list of comma-separate pairs, or implicitly using a geometric description. The order of the parameters may also be reversed with the `order` parameter. An example specification:

```
points    = 60,-40:60,-160:150,-160:180,-100:150,-40
order     = reverse  // default is "normal"
repeat    = 3        // default is 0
```

A waypoint behavior with this specification will traverse the five points in reverse order (150, −40 first) four times (one initial cycle and then repeated three times) before completing. If there is a syntactic error in this specification at helm start-up, an output error will be generated and the helm will not continue to launch. If the syntactic error is passed as part of a dynamic update (see Section 7.2.2), the change in waypoints will be ignored and the a warning posted to the `BHV_WARNING` variable. See Section 12 for more methods for specifying sets of waypoints. The behavior can be set to repeat its waypoints indefinitely by setting `repeat="forever"`.

8.1.3 The `capture_radius` and `nonmonotonic_radius` Parameters

The `capture_radius` parameter specifies the distance to a given waypoint the vehicle must be before it is considered to have arrived at or achieved that waypoint. It is the inner radius around the points in Figure 22. The non-monotonic radius or `nm_radius` parameter specifies an alternative criteria for achieving a waypoint.

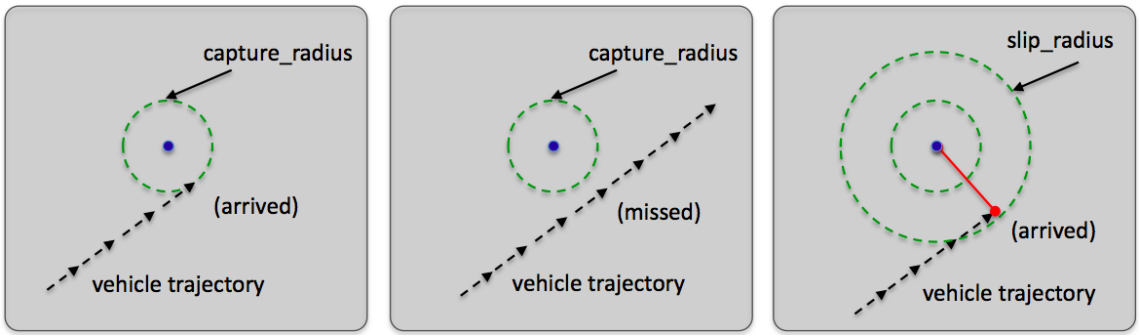


Figure 23: **The capture radius and non-monotonic radius:** (a) a successful waypoint arrival by achieving proximity less than the capture radius. (b) a missed waypoint likely resulting in the vehicle looping back to try again. (c) a missed waypoint but arrival declared anyway when the distance to the waypoint begins to increase and the vehicle is within the non-monotonic radius.

As the vehicle progresses toward a waypoint, the sequence of measured distances to the waypoint decreases monotonically. The sequence becomes non-monotonic when it hits its waypoint or when there is a near-miss of the waypoint capture radius. The `nm_radius`, is a capture radius distance within which a detection of increasing distances to the waypoint is interpreted as a waypoint arrival. This distance would have to be larger than the capture radius to have any effect. As a rule of thumb, a distance of twice the capture radius is practical. The idea is shown in Figure 23. The

behavior keeps a running tally of hits achieved with the capture radius and those achieved with the non-monotonic radius. These tallies are reported in a status message described in Section 8.1.5 below.

8.1.4 Track-line Following using the `lead` Parameter

By default the waypoint behavior will output a preference for the heading that is directly toward the next waypoint. By setting the `lead` parameter, the behavior will instead output a preference for the heading that keeps the vehicle closer to the track-line, or the line between the previous waypoint and the waypoint currently being driven to.

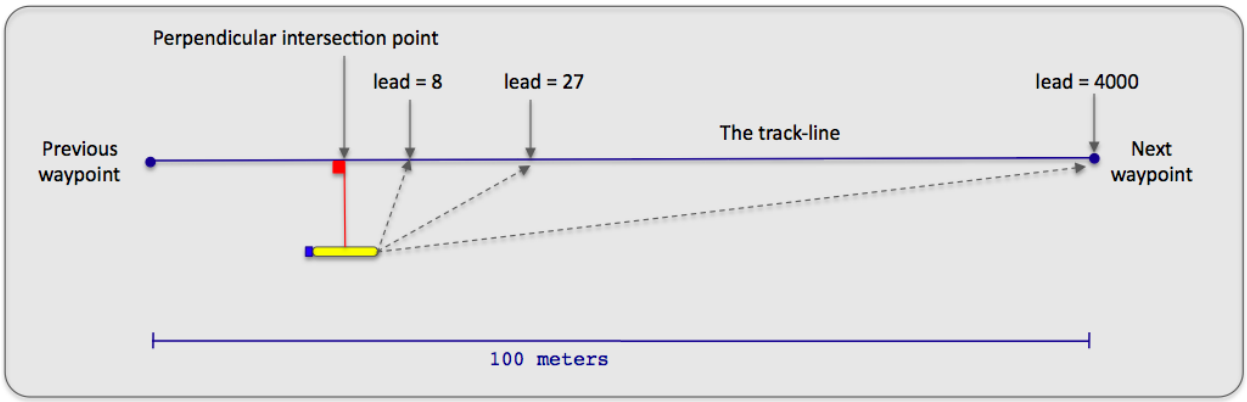


Figure 24: **The track-line mode:** When in track line mode, the vehicle steers toward a point o the track line rather than simply toward the next waypoint. The steering-point is determined by the `lead` parameter. This is the distance from the perpendicular intersection point toward the next waypoint.

The distance specified by the `lead` parameter is based on the perpendicular intersection point on the track-line. This is the point that would make a perpendicular line to the track-line if the other point determining the perpendicular line were the current position of the vehicle. The distance specified by the `lead` parameter is the distance from the perpendicular intersection point toward the next waypoint, and defines an imaginary point on the track-line. The behavior outputs a heading preference based on this imaginary steering point. If the lead distance is greater than the distance to the next waypoint along the track-line, the imaginary steering point is simply the next waypoint.

Normally, when trackline following is enabled, it is enabled only between the first waypoint and all succeess waypoints. When the parameter `lead_on_start` is set to true, trackline following is attempted even to the first waypoint, by defining a trackline from the vehicle's present position when the behavior first enters the running mode.

If the `lead` parameter is enabled, it may be optionally used in conjuction with the `lead_damper` parameter. This parameter expresses a distance from the trackline in meters. When the vehicle is within this distance, the value of the `lead` parameter is stretched out toward the next waypoint to soften, or dampen, the approach to the trackline and reduce overshooting the trackline.

8.1.5 Variables Published by the BHV_Waypoint Behavior

The waypoint behavior publishes five variables for monitoring the performance of the behavior as it progresses: WPT_STAT, WPT_INDEX, CYCLE_INDEX, VIEW_POINT, VIEW_SEGLIST. The WPT_STAT contains information identifying the vehicle, the index of the current waypoint, the type of hits recorded for each waypoint, the distance to the current waypoint, and the estimated time of arrival to the current waypoint. Example output:

```
WPT_STAT = "vname=alpha,behavior=traverse1,index=0,hits=10/11,dist=43,eta=23"
```

The "hits=10/11" component in the above example indicates that, of the 11 waypoint arrivals achieved so far, 10 of them were achieved by meeting the capture radius criteria, and one of them was achieved by meeting the nonmonotonic radius criteria.

The WPT_INDEX variable simply publishes the index of the current waypoint. This is a bit redundant since this information is also in the WPT_STAT posting, but this variable is logged as a numerical variable, not a string, and facilitates the plotting of the index value as a step function in post mission analysis tools. The CYCLE_INDEX variable publishes the number of times the behavior has traversed the entire set of waypoints. The behavior may be configured to post the information in these three variables using alternative variables of the user's liking:

```
post_mapping  = WPT_STAT, MY_WPT_STATUS_VAR
post_mapping  = WPT_INDEX, MY_WPT_INDEX_VAR
post_mapping  = CYCLE_INDEX, MY_CYCLE_INDEX
```

or, to suppress the reports completely:

```
post_mapping  = WPT_STAT, SILENT
post_mapping  = WPT_INDEX, SILENT
post_mapping  = CYCLE_INDEX, SILENT
```

Further posts to the MOOSDB can be configured to be made at the end of each cycle, that is, after reaching the last waypoint. Normally, if the repeat parameter remains at its default value of zero, then the end of a cycle and completing are identical and endflags can be used to post the desired information. However, when the behavior is configured to repeat the set of waypoints one or more times before completed, the cycleflag parameter may be used to post one or more variable-value pairs at the end of each cycle. Likewise, if the repeat parameter is zero, but the behavior is set with perpetual=true, the cycle flags will posted each new time that the behavior completes.

The VIEW_POINT and VIEW_SEGLIST variables provide information consumable by a GUI application such as pMarineViewer or alogview for rendering the set of waypoints traversed by the behavior (VIEW_SEGLIST) and the behavior's next waypoint (VIEW_POINT). These two variables are responsible for the visual output in the Alpha Example Mission in Section 4 in Figure 6 on page 39.

The Objective Function Produced by the Waypoint Behavior

The waypoint behavior produces a new objective function, at each iteration, over the variables speed and course/heading. The behavior can be configured to generate this objective function in one of two forms, either by coupling two independent one-variable functions, or by generating a single coupled function directly. The functions rendered in Figure 25 are built in the first manner.

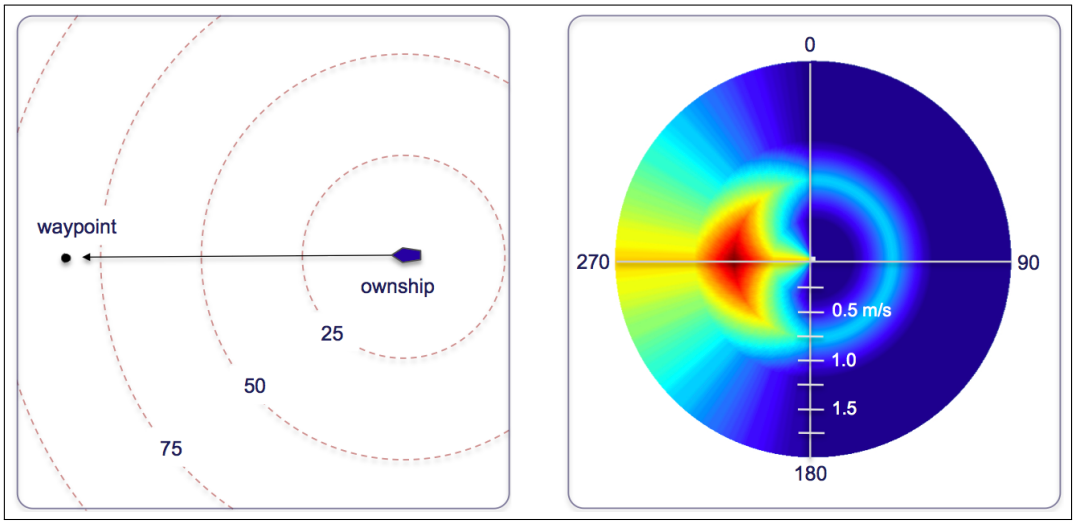


Figure 25: **A waypoint objective function:** The objective function produced by the waypoint behavior is defined over possible heading and speed values. Depicted here is an objective function favoring maneuvers to a waypoint 270 degrees from the current vehicle position and favoring speeds closer to the mid-range of capable vehicle speeds. Higher speeds are represented farther radially out from the center.

8.1.6 Further Clarification on the `repeat` vs. `perpetual` parameter

It's worth clarifying the difference in usage and effect between the `repeat` parameter, which is specific to the Waypoint behavior, and the `perpetual` parameter which is defined for all helm behaviors. Normally when a behavior completes, it is entered into the completed stated, never again to be called upon by the helm. See Section 6.5.3 for more on behavior run states. It is up to the behavior implementor to decide what it means to be complete, and the implentor typically invokes the `setComplete()` function from within the code. See Section 7.5.1 for more on this function. In the case of the Waypoint behavior, completion by default occurs when the vehicle has hit all its waypoints. By setting `perpetual=true` the behavior, upon hitting all its waypoints, still invokes the `setComplete()` function, which causes it to post its endflags, but it does not enter the completed state. This feature is used, for example, in the Alpha example mission to allow the behavior to repeatedly return to the start point and be re-deployed, and later return to its start point again using the same Waypoint behavior.

The `repeat` parameter is used to change the criteria for completion. Whereas the normal criteria for completion is hitting all waypoints once, using `repeat=N` changes the criteria to be hitting all waypoints `N+1` times. A few rules of thumb may be helpful in keeping things straight:

- When `perpetual=false`, the default setting, the behavior will permanently enter the completed state once it has hit all its waypoints.
- When `perpetual=true` and the behavior does not post endflags leading to its run conditions being unsatisfied, the behavior will repeat its waypoints indefinitely.
- When `perpetual=true` and it does indeed post endflags that lead to its run conditions being unsatisfied, it will remain in a running state until all its waypoints are hit. If the `repeat` parameter is used, it won't post its endflags until it has repeated all waypoints the specified

number of times. During the course of traversal, the `cycleflag` posts will be made each time it has completed the set of waypoints. Upon completion, it will post its endflags and reset its cycle counter.

- By setting `repeat=N`, where `N>0`, the `perpetual` parameter is automatically set to `true`.

8.2 BHV_OpRegion

This behavior provides four different types of safety functionality, (a) a boundary box given by a convex polygon in the x-y or lat-lon plane, (b) an overall timeout, (c) a depth limit, (d) an altitude limit. The behavior does not produce an objective function to influence the vehicle to avoid violating these safety constraints. This behavior merely monitors the constraints and posts an error which results in the posting of all-stop commands, and puts the vehicle into the PARK state.

8.2.1 Brief Overview of Configuration Parameters and Variables Published

The configuration parameters and variables published collectively define the interface for the behavior. A more detailed description of usage is provided other parts of this section, and in Table 32 on page 267.

Configuration Parameters for the BHV_OpRegion Behavior

The following are the parameters for this behavior, in addition to the configuration parameters defined for all behaviors, described in Section 7.2 beginning on page 81.

Parameter	Description BHV_OpRegion
MAX_TIME	The max allowable time in seconds. Section 8.2.3.
MAX_DEPTH	The max allowable depth in meters. Section 8.2.3.
MIN_ALTITUDE	The min allowable altitude in meters. Section 8.2.3.
POLYGON	The lat-lon area the vehicle is restricted to stay within. Section 8.2.2.
TRIGGER_ENTRY_TIME	The time required for the vehicle to have been within the polygon region before triggering the polygon requirement. Section 8.2.2.
TRIGGER_EXIT_TIME	The time required to have been outside the polygon before declaring a polygon containment failure. Section 8.2.2.
VISUAL_HINTS	Hints for visual properties in variables posted intended for rendering.

Table 7: Configuration paramters for the BHV_OpRegion behavior.

Variables Published by the BHV_OpRegion Behavior

The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be supressed or changed to a different variable name using the post_mapping configuration parameter described in Section 7.2.1 on page 82.

MOOS Variable	Description
OPREG_TRAJECTORY_PERIM_DIST:	Distance, in meters, to the perimeter on the current trajectory.
OPREG_TRAJECTORY_PERIM_ETA:	Time to the perimiter on the current speed and trajectory.
OPREG_ABSOLUTE_PERIM_DIST:	Distance, in meters, to the op-region in any direction.
OPREG_ABSOLUTE_PERIM_ETA:	Time to the op-region perimeter at current speed in any direction.
OPREG_TIME_REMAINING:	Time, in seconds, until the max_time upper bound is exceeded.
VIEW_POLYGON:	A visual cue for rendering the polygon containment region.

Table 8: Variables posted by the BHV_OpRegion behavior.

The following are some examples:

```
 OPREG_TRAJECTORY_PERIM_DIST = 498.4
  OPREG_TRAJECTORY_PERIM_ETA = 875.0
   OPREG_ABSOLUTE_PERIM_DIST = 17.9
    OPREG_ABSOLUTE_PERIM_ETA = 30.2
        OPREG_TIME_REMAINING = 1134.0
                VIEW_POLYGON = edge_size,0.0:vertex_size,0.0:0,-50:0,-150:150,-150:150,-50
```

8.2.2 Safety Checking Applied to an Operation Region

One safety check performed by the OpRegion behavior is to ensure that the vehicle remains in an operation region defined by a convex polygon in the x-y plane.

POLYGON: A colon separated list of x,y pairs given as points in space, typically meters. A pair given by "label,string" can associate an optional label with the point list. *The collection of points must be a convex polygon.* A check for convexity is done upon helm/behavior start-up. Behavior initialization will fail if it is not convex. If no polygon is provided, no X,Y checks are made.

TRIGGER_ENTRY_TIME: The amount of time required for the vehicle to have been within the polygon containment region before triggering the polygon containment requirement. This is useful when launching vehicles from a dock structure such as the MIT Sailing Pavilion. The default setting is zero meaning the polygon containment requirement is active immediately.

TRIGGER_EXIT_TIME: The amount of time required to have been outside the polygon containment region before declaring a polygon containment failure. This is useful if the vehicle NAV_X and NAV_Y position is based on a sensor without outlier detection. The kayaks, for example, are often relying solely on GPS which occasionally emits an outlier well out of the containment region. By setting this value high enough, outliers are ignored. Each time a recorded position is contained within the polygon region, the clock is set to zero. The default setting is zero, meaning the very first detection outside the polygon will result in a polygon containment error.

8.2.3 Safety Limits on Operation Time, Vehicle Depth, and Vehicle Altitude

MAX_TIME: The maximum allowable time (in seconds) that the helm is allowed to run. The clock starts when the pHelmIvP process first takes control, i.e., enters the DRIVE state. If no maximum time is specified, then no time checks are made.

MAX_DEPTH: The maximum allowable depth of the vehicle (in meters). If no depth is provided, no depth checks are made.

MIN_ALTITUDE: The minimum allowable altitude of the vehicle (in meters). If no altitude is provided, no altitude checks are made.

8.2.4 Variables Published by the BHV_OpRegion Behavior

The behavior also produces a set of status variables regarding the vehicle position with respect to the containment region. Since a violation of this constraint results in a vehicle full-stop and the helm relinquishing control, other behaviors or MOOS processes may want to take measures to avoid it. These status variables provide information on the position and estimated time between the vehicle and the perimeter, based both on the absolute position as well as the current vehicle trajectory. See Figure 26.

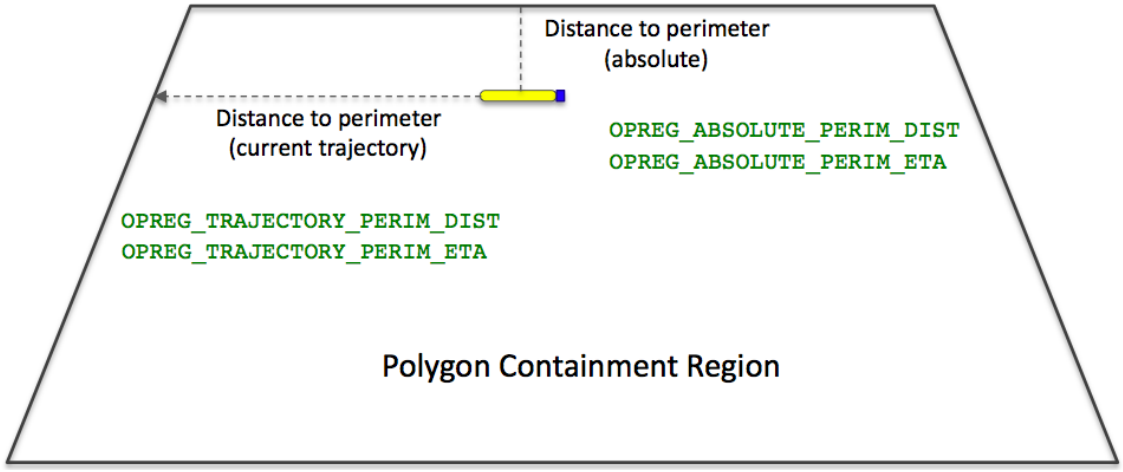


Figure 26: **The OpRegion polygon and status variables:** The OpRegion behavior publishes information regarding its estimated distance and time of arrival (ETA) to the perimeter of the polygon containment region. It publishes two sets of information; one based on the current trajectory and the other based on the absolute distance to the perimeter at top vehicle speed.

The four variables produced by the behavior (and posted to the MOOSDB by the Helm) are:

OPREG_TRAJECTORY_PERIM_DIST: The distance (in meters) between the current vehicle position to the perimeter of the polygon containment region (given by the POLYGON parameter), based on the vehicle remaining on the current trajectory.

OPREG_TRAJECTORY_PERIM_ETA: The amount of time (in seconds) needed for the vehicle to reach the perimeter of the polygon containment region (given by the POLYGON parameter), based on the vehicle remaining on the current trajectory.

OPREG_ABSOLUTE_PERIM_DIST: The distance (in meters) between the current vehicle position to the perimeter of the polygon containment region (given by the POLYGON parameter), regardless of the current vehicle trajectory.

`OPREG_ABSOLUTE_PERIM_ETA:` The amount of time (in seconds) needed for the vehicle to reach the perimeter of the polygon containment region (given by the `POLYGON` parameter), regardless of the current vehicle trajectory. Calculated on the maximum vehicle speed.

8.3 BHV_Loiter

A behavior for transiting to and repeatedly traversing a set of waypoints. A similar effect can be achieved with the BHV_Waypoint behavior but this behavior assumes a set of waypoints forming a convex polygon to exploit certain useful algorithms discussed below. It also utilizes the non-monotonic arrival criteria use in the BHV_Waypoint behavior to avoid loop-backs upon waypoint near-misses. It also robustly handles dynamic exit and re-entry modes when or if the vehicle diverges from the loiter region due to external events. And it is dynamically reconfigurable to allow a mission control module to repeatedly reassign the vehicle to different loiter regions by using a single persistent instance of the behavior.

8.3.1 Brief Overview of Configuration Parameters and Variables Published

The configuration parameters and variables published collectively define the interface for the behavior. A more detailed description of usage is provided other parts of this section, and in Table 33 on page 268.

Configuration Parameters for the BHV_Loiter Behavior

The following are the parameters for this behavior, in addition to the configuration parameters defined for all behaviors, described in Section 7.2 beginning on page 81.

Parameter	Description BHV_Loiter
ACQUIRE_DIST:	Distance to polygon outside which the behavior will be in "acquire" mode.
CAPTURE_RADIUS:	The radius tolerance, in meters, for satisfying the arrival at a point.
CENTER_ACTIVATE:	If `true`, center of polygon is set to present position when behavior activates.
CENTER_ASSIGN:	An x,y pair, in meters, indicating a (new) center of the loiter polygon.
CLOCKWISE:	If `true`, loitering is done clockwise (the default setting).
NM_RADIUS	A depricated alias for SLIP_RADIUS.
POLYGON:	Polygon about which the vehicle traverse and loiter.
POST_SUFFIX:	A string to add as a suffix to variables posted by this behaviors.
RADIUS:	An alias for CAPTURE_RADIUS.
SLIP_RADIUS:	An "outer" capture radius. Arrival declared when the vehicle is in this range and the distance to the point begins to increase.
SPEED:	Speed in meters per second.
SPIRAL_FACTOR:	The degree of spiraling when activated at th ecenter of the polygon.
VISUAL_HINTS:	Hints for visual properties in variables posted intended for rendering.
XCENTER_ASSIGN:	A x-value, in meters, indicating a (new) x-position of the loiter polygon.
YCENTER_ASSIGN:	A y-value, in meters, indicating a (new) y-position of the loiter polygon.

Table 9: Configuration parameters for the BHV_Loiter behavior.

Variables Published by the BHV_Loiter Behavior

The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be supressed or changed to a

different variable name using the post_mapping configuration parameter described in Section 7.2.1 on page 82.

MOOS Variable	Description
LOITER_ACQUIRE	Posts 1 when in the "acquire" mode, 0 otherwise.
LOITER_DIST_TO_POLY	Current distance, in meters, to the loiter polygon.
LOITER_ETA_TO_POLY	Estimated time of arrival to the polygon at present speed and trajectory.
LOITER_INDEX	The index of the vertex in the loiter polygon currently heading toward.
LOITER_MODE	A string indicating details of the acquire mode, e.g., "acquiring_external".
LOITER_REPORT	A status string with current mode, current vertex, and prior arrivals.
VIEW_POINT:	A visual cue for rendering the next point in the loiter polygon.
VIEW_POLYGON:	A visual cue for rendering the loiter polygon.

The following are some examples:

```
      LOITER_REPORT = index=5,capture_hits=51,nonmono_hits=0,acquire_mode=false
       LOITER_INDEX = 5
    LOITER_ACQUIRE = 1
LOITER_DIST_TO_POLY = 2.2
 LOITER_ETA_TO_POLY = 294.4
       LOITER_MODE = stable
      VIEW_POLYGON = edge_size,0.0:vertex_size,0.0:0,-50:0,-150:150,-150:150,-50
```

8.3.2 Setting and Altering the Loiter Region

The Loiter behavior is configured with a *loiter region*, defined by a convex polygon. The behavior will influence the vehicle to repeatedly traverse the set of points on the polygon. The polygon is specified typically in the behavior configuration block. Any string that properly defines a convex polygon (see Section 12) is acceptable. The following two configuration lines, for example, will result in the same polygon.

```
polygon = 20,-40:40,-75:20,-110:-20,-110:-40,-75:-20,-40:label,Lima

polygon = format=radial, x=0, y=-75, radius=40, pts=6, snap=1, label=Lima
```

Each would produce the polygon shown in Figure 27:

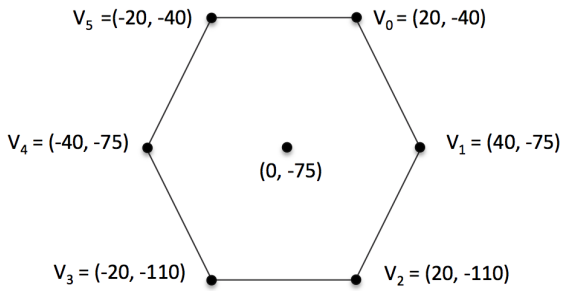


Figure 27: A typical loiter polygon with six vertices.

The shape and position polygon may be alterered dynamically (after the helm is launched and running) in one of two manners by either specifying new parameters explicitly, or by tying the loiter position to the vehicle position when the loiter behavior enters the running behavior mode.

Updating the Polygon Position with the UPDATES Variable

In the first case, altering the polygon parameter dynamically is accomplished by using the standard UPDATES parameter described in Section 7.2.2. For example, the behavior may be configured to receive new updates by adding the following line to its configuration block:

```
UPDATES = NEW_CONFIG
```

Its position may be then moved 75 meters North by posting the following to the MOOSDB:

```
NEW_CONFIG = "polygon = format=radial, x=0, y=0, radius=40, pts=6, snap=1, label=Lima"
```

Alternatively, if one wants to simply move the polygon to a new (x,y) position, the Loiter behavior also implements the CENTER_ASSIGN configuration parameter. The same shift as above can be made without the source making the post having to know anything about the other parameters of the polygon:

```
NEW_CONFIG = "center_assign = 0,0"
```

Likewise, if one simply wants to move the polygon in either the x or y direction without knowing anything about the position in the other direction, the Loiter behavior implements the XCENTER_ASSIGN and YCENTER_ASSIGN parameters. Thus the above shift could also be accomplished without the source making the post knowing anything about the current x position of the polygon:

```
NEW_CONFIG = "ycenter_assign = 0"
```

Updating the Polygon Position Using the Current Vehicle Position

The Loiter behavior may also be configured to reset the (x, y) center position of the loiter polygon to the present position of the vehicle at the very the moment the behavior transitions from the idle to the running state. See Section 6.5.3 for more on behavior run states. This configuration is declared with the following line in the behavior configuration block:

```
CENTER_ACTIVATE = true
```

This setting is useful if one wants to, for example, send a command to the vehicle to exit some other mission mode and simply loiter at its present location until further notice. Of course this configuration as well, may also be dynamically toggled through a variable specified in the UPDATES parameter.

8.3.3 Setting the Loiter Direction

The user may configure the direction the vehicle traverses the loiter polygon by setting the parameter:

```
clockwise = <value>
```

The possible case insensitive settings for `<value>` are `"true"`, `"false"`, `"best"`. In the first two cases, the directions are explicitly set and will not vary regardless of the pose of the vehicle with respect to the polygon. In the case where `clockwise=best`, the direction is re-evaluated once, whenever the behavior run state transitions from idle to running. Thus the direction depends on the pose relative position to the polygon at that particular point in time. This is shown in the lower case in Figure 28 below.

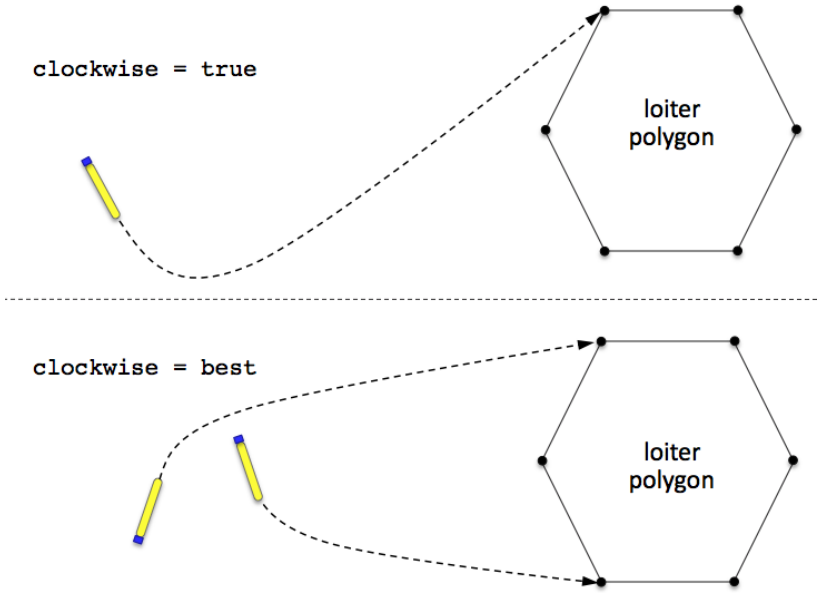


Figure 28: **The Loiter Direction.** The polygon traversal direction is determined by the `clockwise` configuration parameter. It may be explicitly set as shown on the top, or determined at run time when the behavior becomes non-idle as shown on the bottom.

Regardless of the prevailing direction, as the vehicle is transiting to the loiter polygon, the behavior will influence the vehicle toward the vertex that allows for the smoothest entry, given the chosen direction. If the polygon were a perfect circle, the vehicle would approach on one of the two tangent lines.

8.3.4 The Loiter Acquisition Mode

When the Loiter behavior is running (non-idle), it behaves differently depending on where the vehicle is in relation to the loiter polygon. The behavior has a notion of (a) when it is progressing in a "stable" manner around the polygon and (b) when it is trying to move the vehicle back to (a). The difference between the two is solely determined by whether or not the vehicle is within

some *acquire distance* from the loiter polygon. This distance is set with the behavior configuration parameter:

```
ACQUIRE_DIST = <distance>
```

The `<distance>` component must simply be a non-negative value. A value of zero should work fine, but essentially disables some useful ways in which the behavior may be coordinated with other parts of the mission. This relationship is shown in Figure 29.

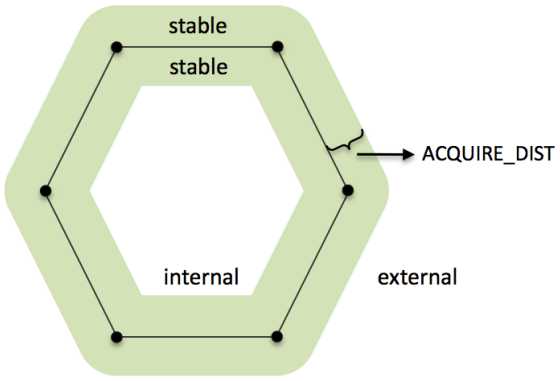


Figure 29: **The Loiter Polygon Zones:** The Loiter behavior regards itself to be in one of three distinct possible zones relative to the polygon boundary - *stable*, *internal*, *external*, depending on the setting of the `acquire_dist` parameter as shown.

When the vehicle is *not* within the stable zone, it is trying to acquire the polygon in one of four distint manners, depending on whether (a) the vehicle is internal or external and (b) whether it is "acquiring" the polygon for the first time, or whether it is "recovering" from having drifted out of the stable zone. The idea is depicted in Figure 30.

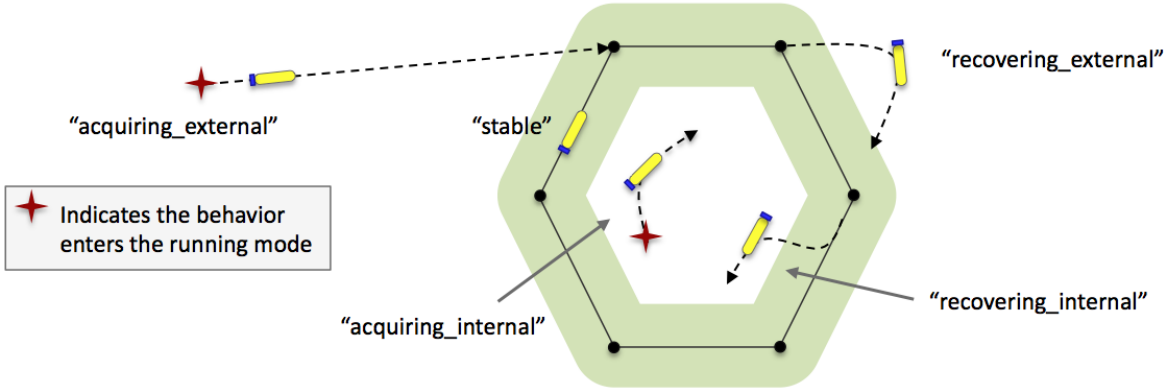


Figure 30: **Loiter Modes**: The behavior is one of five possible modes, *stable*, *acquiring_external*, *recovering_external*, *recovering_internal*, or *acquiring_internal*.

The current loiter mode is published by the behavior in the MOOS variable LOITER_MODE. As with any MOOS variable published by a behavior, this may be remapped to a different variable of the users liking with the POST_MAPPING configuration parameter, described on page 82.

When the behavior is in any loiter mode other than the stable mode, it recalculates on each iteration which vertex it should be heading toward, the *approach vertex*. In the case of an external approach, the chosen vertex should remain steady unless there are external forces such as wind or current, or if the vehicle changes its aspect to the polygon significantly as it is executing a turn. In the case of an internal approach, the approach vertex will likely change during the approach, outward toward the polygon boundary, creating a pseudo outward spiral trajectory. Note that re-evaluating the approach vertex is not the same as re-evaluating the traversal direction of the polygon. The latter is only re-evaluated dynamically if the behavior is configured with clockwise=best, and then only when the behavior becomes non-idle.

The circumstance most common for triggering the acquire mode is the initial assignment to the vehicle to loiter at a new given region in the X,Y plane. This assignment *could* occur while the vehicle happens to already be within the polygon for a number of reasons. Furthermore, the vehicle could be driven off the polygon loiter trajectory due to environmental (wind or current) forces or the temporary dominance of other vehicle behaviors such as collision avoidance or tracking of another vehicle.

Once the behavior enters the acquire mode, it remains in this mode until arriving at the first waypoint (defined by the arrival and non-monotonic radii settings), after which it switches to normal mode until the acquire mode is re-triggered or the behavior run conditions are no longer met. There is currently no "complete" condition for this behavior other than a time-out which is defined for all behaviors.

8.3.5 Parameters

POLYGON: A colon separated list of comma-separated x,y pairs indicating points in 2D space. Units are in in meters. Unlike the waypoint behavior, these points must describe a convex polygon; if the convexity condition fails the behavior will not instantiate. As an alternative to listing a sequence of points, a orbit-style polygon can be given by four values (1),(2) the x and y position, (3) the radius in meters, and (4) the number of points on the circle. This specification is denoted with the "radial" tag as follows "radial:50,50,200,16".

SPEED: The desired speed, in meters/second, at which the vehicle travels through the points.

RADIUS: The radius tolerance, in meters, for satisfying the arrival at a waypoint. As soon as the vehicle is within this distance to the waypoint the waypoint behavior begins operating on the next waypoint in the sequence, or completes and posts its endflags if there are no more waypoints.

SLIP_RADIUS: As the vehicle progresses toward a waypoint, the sequence of measured distances to the waypoint decreases monotonically. The sequence becomes non-monotonic when it hits its waypoint or when there is a near-miss of the waypoint arrival radius. The SLIP_RADIUS, a.k.a, the NM_RADIUS, short for *non-monotonic radius* is an arrival radius distance within which a detection of increasing distances to the waypoint is interpreted as a waypoint arrival. This distance would

have to be larger than the arrival radius to have any effect (see Figure 23). As a rule of thumb, a distance of twice the arrival radius is practical.

`CLOCKWISE:` If "true", the behavior will influence the vehicle in a clockwise direction around the polygon. Values are case insensitive, but must spell either true or false. The default is true.

`ACQUIRE_DIST:` The distance in meters between the vehicle and the polygon that will trigger the vehicle to return to *acquire* mode. This notion applies to the case where the vehicle is both inside and outside the polygon. (The re-acquire algorithms are different however.)

`POST_SUFFIX:` This string will be added as a suffix to each of the status variables posted by the behavior (`LOITER_REPORT`, `LOITER_INDEX`, `LOITER_ACQUIRE`, `LOITER_DIST2POLY`). By default, the suffix is the empty string and the variables will be posted as above. When multiple Loiter behaviors are configured in the helm it may help to distinguish the posted variabes by a suffix. A given suffix of `"FOO"` would result in the posting of `LOITER_INDEX_FOO` for example. The extra `'_'` character is inserted automatically.

8.4 BHV_PeriodicSpeed

This behavior will periodically influence the speed of the vehicle while remaining neutral at other times. The timing is specified by a given period in which the influence is on (busy), and a period specifying when the influence if off (lazy), as depicted in Figure 31.

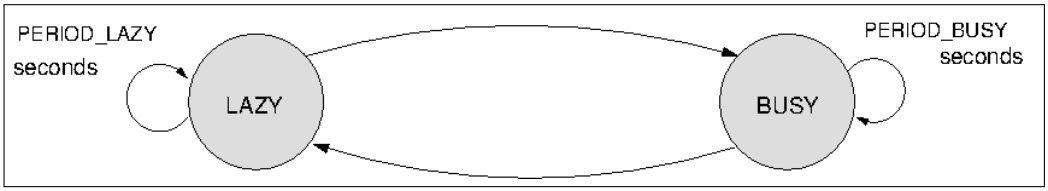


Figure 31: **Busy and Lazy Modes:** In the busy mode the behavior will produce an objective function defined over speed that will potentially influence the speed of the vehicle. In the lazy mode, it simply will not produce an objective function.

It was conceived for use on an AUV equipped with an acoustic modem to periodically slow the vehicle to reduce self-noise and reduce communication difficulty. One can also specify a flag (a MOOS variable and value) to be posted at the start of the period to prompt an outside action such as the start of communication attempts.

8.4.1 Brief Overview of Configuration Parameters and Variables Published

The configuration parameters and variables published collectively define the interface for the behavior. A more detailed description of usage is provided other parts of this section, and in Table 34 on page 269.

8.4.2 Parameters of the BHV_PeriodicSpeed Behavior

The following are the parameters for this behavior, in addition to the configuration parameters defined for all behaviors, described in Section 7.2 beginning on page 81.

Parameter	Description BHV_PeriodicSpeed
BASEWIDTH:	The width of the ZAIC basewidth, in m/sec, in the IvP function.
INITIALLY_BUSY:	If true the initial state is busy. The default is false.
PEAKWIDTH:	The width of the ZAIC peakwidth, in m/sec in the IvP function.
PERIOD_BUSY:	The duration of the busy period, in seconds.
PERIOD_LAZY:	The duration of the lazy period, in seconds.
PERIOD_SPEED:	The desired speed, in m/sec, in the IvP function.
RESET_UPON_RUNNING:	Initial conditions reset upon entering the running state. Default is true.
SUMMIT_DELTA:	The extent of the ZAIC summit delta in the IvP function.

8.4.3 State Transition Policy and Initial Condition Parameters

The behavior alternates between one of two modes, the busy mode or the lazy mode. In the former, it will produce an objective function over the `speed` decision variable, and in the latter mode it will simply refrain from producing the objective function. Note that these modes are different from the general behavior run states described in Section 6.5.3 on page 73. If this behavior is idle, i.e., has not met the conditions for being in the running state, it will not generate an objective function regardless of whether it is in the busy or lazy mode.

When the behavior enters the busy mode, it resets a timer which counts down from `PERIOD_BUSY` seconds, after which it enters the lazy mode. When it enters the lazy mode, it resets a timer which counts down from `PERIOD_LAZY` seconds, after which it goes back to the busy mode. By default the behavior is initially in the lazy mode, but it can be configured in the opposite manner by setting `INITIALLY_BUSY` to `true`.

By default, when `RESET_UPON_RUNNING` is `true`, each time the behavior enters the running state, the busy/lazy mode is set to its initial value, and all timers are reset to their initial values. This is true regardless of whether it is entering the running state upon initial helm engagement, or due to transitioning from the idle state. This can be changed by setting `RESET_UPON_RUNNING` to `false`. In this case the timers are counting down immediately upon helm engagement and continue to do so regardless of the behavior run state.

8.4.4 Variables Published by the BHV_PeriodicSpeed Behavior

This behavior publishes three variables for monitoring and logging performance - `PS_PENDING_BUSY`, `PS_PENDING_LAZY`, and `PS_BUSY_COUNT`.

The `PS_PENDING_BUSY` variable publishes the amount of time in seconds until the behavior reaches the busy mode. During the busy mode this value is zero, and it resets to the value given by the `PERIOD_LAZY` parameter upon transitioning into the lazy mode. The `PS_PENDING_LAZY` variable publishes the amount of time in seconds until the behavior reaches the lazy mode. During the lazy mode this value is zero, and it resets to the value given by `PERIOD_BUSY` upon transitioning back into the busy mode. To reduce posting volume, the values posted will be rounded to the nearest second until less than one second remains, after which fractions are posted. The `PS_BUSY_COUNT` is posted by the behavior each time it enters the busy mode. The value is an integer indicating the number of times it has entered the busy mode, posting zero initially.

8.5 BHV_PeriodicSurface

This behavior will periodically influence the depth and speed of the vehicle while remaining neutral at other times. The purpose is to bring the vehicle to the surface periodically to achieve some event specified by the user, typically the receipt of a GPS fix. Once this event is achieved, the behavior resets its internal clock to a given period length and will remain idle until a clock time-out occurs. The behavior can be in one of four states as described in Figure 32 below.

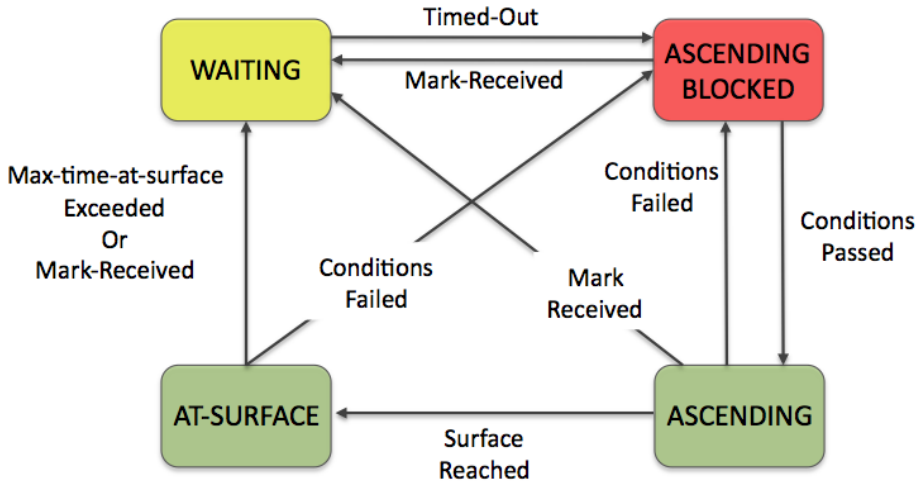


Figure 32: Possible modes of the PeriodicSurface behavior.

In the `WAITING` state the behavior is simply waiting for its clock to wind down to zero. The duration is given by the `PERIOD` parameter listed below. The clock is active despite any other run conditions that may apply to the behavior. It is started when the behavior is first instantiated and also when the desired event occurs at the surface. The `ASCENDING_BLOCKED` state indicates that the behavior timer has reached zero, but another run condition has not been met. This is to prevent the behavior from trying to surface the vehicle when other circumstances override the need to surface. In the `ASCENDING` state, the behavior will produce an objective function over depth and speed to bring the vehicle to the surface. A couple parameters described below can determine the trajectory of the vehicle during ascent. This state can transition back to the `ASCENDING_BLOCKED` state if run conditions become no longer satisfied prior to the vehicle reaching the surface. In the `AT_SURFACE` state the vehicle is at the surface waiting for a specified event.

Parameter	Description BHV_PeriodicSurface
ACOMMS_MARK_VARIABLE	The incoming MOOS variable for resetting the acomms period clock.
ASCENT_GRADE	Manner in which desired speed approaches zero on the approach to the surface.
ASCENT_SPEED	Desired speed of the vehicle during the `ASCENT` mode.
MARK_VARIABLE	The incoming MOOS variable for resetting the period clock.
MAX_TIME_AT_SURFACE	The maximum time, in seconds, the vehicle will wait at the surface.
PERIOD	Duration of the `WAITING` mode.
ZERO_SPEED_DEPTH	The depth, in meters, at which the desired speed becomes zero on ascent.

Table 10: Configuration parameters for the BHV_PeriodicSurface behavior.

PERIOD: The duration of the period, in seconds, during which the behavior will remain in the IDLE_WAITING state.

MARK_VARIABLE: The name of a variable used for indicating when the behavior witnesses the event that would reset the period clock. On each iteration, the variable is checked against its last known value and if different, the clock is reset. The default value for this parameter is GPS_UPDATE_RECEIVED. If this variable is populated by another process with a value indicating the time a GPS fix is obtained, then the mark will occur on each GPS fix. Since the value of this argument names a MOOS variable, it is case sensitive.

PENDING_STATUS_VAR: This variable will be written to with the value of the remaining time on the idle clock, rounded to integer seconds. The default value is PENDING_SURFACE. Since the value of this argument names a MOOS variable, it is case sensitive.

ATSURFACE_STATUS_VAR: This variable will be written to with the number of seconds that the vehicle has been waiting at the surface (for the event indicated by the MARK_VARIABLE). The number of seconds is rounded to the nearest integer and will be zero when the vehicle is not at the surface. The default value is TIME_AT_SURFACE. Since the value of this argument names a MOOS variable, it is case sensitive.

ASCENT_SPEED: This parameter indicates the desired speed (m/s) of the vehicle during the ascent state. If left unspecified, the ascent speed will be equal to the current noted speed at moment it transitions into the ascent state.

ASCENT_GRADE: This parameter indicates the manner in which the ascent speed approaches zero as the vehicle progresses toward the ZERO_SPEED_DEPTH. It has four legal values: *fullspeed*, *linear*, *quadratic*, and *quasi*. In all four cases, the initial speed is determined by the parameter ASCENT_SPEED, and the desired speed will be zero once the ZERO_SPEED_DEPTH has been achieved. The four settings determine the manner of slowing to zero speed during the ascent. The *fullspeed* setting indicates that desired speed should remain constant through the ascent right up to the instant the vehicle achieves ZERO_SPEED_DEPTH. For the other three settings the speed reduction is relative to the starting depth (the depth noted at the outset of the ascent state) and the ZERO_SPEED_DEPTH. With the *linear* setting, the speed reduction is linear. With the *quadratic* setting, the speed reduction is quadratic (quicker initial speed reduction). With the *quasi* setting the speed reduction is between linear and quadratic. The value passed to this parameter is not case sensitive.

ZERO_SPEED_DEPTH: The depth (in meters) during the ascent state at which the desired speed becomes zero, and presumably further ascent is achieved through positive buoyancy.

MAX_TIME_AT_SURFACE: The maximum time (in seconds) spent in the AT_SURFACE state, waiting for the event indicated by the MARK_VARIABLE, before the behavior transitions into the IDLE state.

8.6 BHV_ConstantDepth

This behavior will drive the vehicle at a specified depth. This behavior merely expresses a preference for a particular depth. If other behaviors also have a depth preference, coordination/compromise will take place through the multi-objective optimization process. The following parameters are defined for this behavior:

Parameter	Description BHV_ConstantDepth
BASEWIDTH	The width of the base, in meters, in the produced ZAIC-style IvP function. See Figure 21.
DEPTH	The desired depth of the vehicle, in meters.
DURATION	Behavior duration, in seconds. Mandatory configuration for this behavior.
PEAKWIDTH	The width of the peak, in meters, in the produced ZAIC-style IvP function. See Figure 21.
SUMMITDELTA	The height of the summit delta parameter in the produced ZAIC-style IvP function. See Figure 21.
DEPTH_MISMATCH_VAR	Name of the MOOS variable indicating the present delta between the desired depth and the current depth. If left unspecified, no posting is made.

Table 11: Configuration parameters for the ConstantDepth behavior.

BASEWIDTH: The width of the base, in meters in the produced objective function. The default is 100. See Figure 21 for more on the basewidth parameter used in the ZAIC tool for building IvP functions.

DEPTH: The desired depth in meters. The default is 0.

DURATION: This is a parameter defined for all general behaviors, but for this behavior, specification is mandatory for safety reasons. The default if not specified is 0 seconds which will result in the behavior completing immediately. If no duration limit is desired, e.g., if the behavior is tied to another behavior or event via condition variables, then setting "duration = no-time-limit" will result in no time duration checks for this behavior.

PEAKWIDTH: The width of the peak in meters in the produced objective function. The default is 3. See Figure 21 for more on the peak parameter used in the ZAIC tool for building IvP functions.

SUMMITDELTA: The width of the base, in meters in the produced objective function. The default is 50. See Figure 21 for more on the summitdelta parameter used in the ZAIC tool for building IvP functions.

8.7 BHV_ConstantHeading

This behavior will drive the vehicle at a specified ~~depth~~. This behavior merely expresses a preference for a particular heading. If other behaviors also have a heading preference, coordination/compromise will take place through the multi-objective optimization process. The following parameters are defined for this behavior:

Parameter	Description BHV_ConstantHeading
BASEWIDTH	The width of the base, in degrees, in the produced ZAIC-style IvP function. See Figure 21.
DURATION	Behavior duration, in seconds. Mandatory configuration for this behavior.
HEADING	The desired speed of the vehicle, in degrees (North=0).
PEAKWIDTH	The width of the peak, in degrees, in the produced ZAIC-style IvP function. See Figure 21.
SUMMITDELTA	The height of the summit delta parameter in the produced ZAIC-style IvP function. See Figure 21.
HEADING_MISMATCH_VAR	Name of the MOOS variable indicating the present delta between the desired heading and the current heading. If left unspecified, no posting is made.

Table 12: Configuration parameters for the ConstantHeading behavior.

BASEWIDTH: The width of the base, in degrees in the produced objective function. The default is 170. See Figure 21 for more on the basewidth parameter used in the ZAIC tool for building IvP functions.

DURATION: This is a parameter defined for all general behaviors, but for this behavior, specification is mandatory for safety reasons. The default if not specified is 0 seconds which will result in the behavior completing immediately. If no duration limit is desired, e.g., if the behavior is tied to another behavior or event via condition variables, then setting "duration = no-time-limit" will result in no time duration checks for this behavior.

HEADING: The desired heading in degrees (-180, +180]. The default is 0.

PEAKWIDTH: The width of the peak in degrees in the produced objective function. The default is 10. See Figure 21 for more on the peak parameter used in the ZAIC tool for building IvP functions.

SUMMITDELTA: The width of the base, in meters in the produced objective function. The default is 25. See Figure 21 for more on the summitdelta parameter used in the ZAIC tool for building IvP functions.

8.8 BHV_ConstantSpeed

This behavior will drive the vehicle at a specified speed. This behavior merely expresses a preference for a particular speed. If other behaviors also have a speed preference, coordination/compromise will take place through the multi-objective optimization process. The following parameters are defined for this behavior:

Parameter	Description BHV_ConstantSpeed
BASEWIDTH	The width of the base, in meters per second, in the produced ZAIC-style IvP function. See Figure 21.
DURATION	Behavior duration, in seconds. Mandatory configuration for this behavior.
PEAKWIDTH	The width of the peak, in meters per second, in the produced ZAIC-style IvP function. See Figure 21.
SPEED	The desired speed of the vehicle, in meters per second.
SUMMITDELTA	The height of the summit delta parameter in the produced ZAIC-style IvP function. See Figure 21.
SPEED_MISMATCH_VAR	Name of the MOOS variable indicating the present delta between the desired speed and the current speed. If left unspecified, no posting is made.

Table 13: Configuration parameters for the ConstantSpeed behavior.

SPEED: The desired speed in meters/second. The default is 0.

PEAKWIDTH: The width of the peak in meters/second in the produced objective function. The default is 0. See Figure 21 for more on the peak parameter used in the ZAIC tool for building IvP functions.

BASEWIDTH: The width of the base, in meters/second in the produced objective function. The default is 0.2. See Figure 21 for more on the basewidth parameter used in the ZAIC tool for building IvP functions.

DURATION: This is a parameter defined for all general behaviors, but for this behavior, specification is mandatory for safety reasons. The default if not specified is 0 seconds which will result in the behavior completing immediately. If no duration limit is desired, e.g., if the behavior is tied to another behavior or event via condition variables, then setting "duration = no-time-limit" will result in no time duration checks for this behavior.

SUMMITDELTA: The width of the base, in meters/second in the produced objective function. The default is 0. See Figure 21 for more on the summitdelta parameter used in the ZAIC tool for building IvP functions.

8.9 BHV_GoToDepth

This behavior will drive the vehicle to a sequence of specified depths and duration at each depth. The duration is specified in seconds and reflects the time at depth *after* the vehicle has first achieved that depth, where achieving depth is defined by the `CAPTURE_DELTA` parameter. The behavior subscribes for `NAV_DEPTH` to examine the current vehicle depth against the target depth. If the current depth is within the delta given by `CAPTURE_DELTA`, that depth is considered to have been achieved. The behavior also stores the previous depth from the prior behavior iteration, and if the target depth is between the prior depth and current depth, the depth is considered to be achieved regardless of whether the prior or current depth is actually within the `CAPTURE_DELTA`. This behavior merely expresses a preference for a particular depth. If other behaviors also have a depth preference, coordination/compromise will take place through the multi-objective optimization process. The following parameters are defined for this behavior:

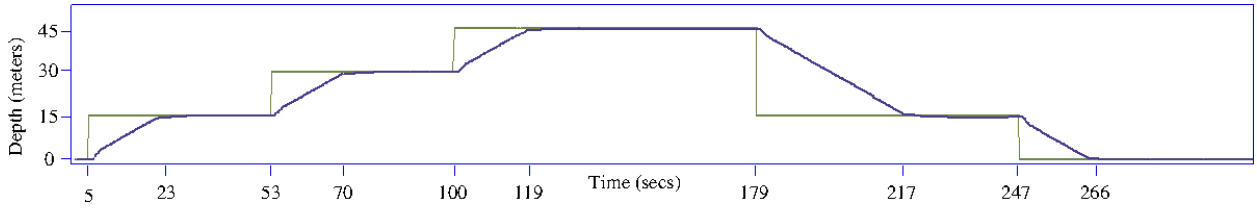


Figure 33: Depth log from simulation with the depth parameters shown in Listing 8. The lighter, step-like line indicates the values of `DESIRED_DEPTH` generated by the helm, and the darker line indicates the recorded depth value of the vehicle. The depth plateaus start from the moment the vehicle achieves depth. For example, the vehicle achieved a depth of 45 meters at 119 seconds and retained that desired depth for another 60 seconds as requested in the configuration shown in Listing 8.

`CAPTURE_DELTA:` The delta depth, in meters, between the current observed depth and the current target depth, below which the behavior will declare the depth to have been achieved. The default value is 1 meter.

`CAPTURE_FLAG:` The name of a MOOS variable incremented each time a target depth level has been achieved. Useful for logfile debugging/analyzing and also allows other behaviors to be conditioned on a depth event. If this behavior is completed in *perpetual* mode, the counter is reset to zero. If the behavior is repeating a set of depths by setting `REPEAT` greater than zero, the counter will continue to increment through evolutions. The default value is the empty string, meaning nothing will be posted. Note the named MOOS variable will automatically have the prefix `"GTD_"` applied.

`DEPTH:` A colon-separated list of comma-separated pairs. Each pair contains a desired depth and a duration at that depth. The duration applies from the point in time that the depth is first achieved. If a time duration is not provided for any pair, it defaults to zero. Thus "depth = 20" is a valid parameter setting.

`REPEAT:` The number of times the vehicle will traverse through the evolution of depths, proceeding to the 1st depth after the nth depth has been hit. The default value is zero.

`PERPETUAL:` If equal to *true*, when the vehicle completes its evolution of depths (perhaps several evolutions if `REPEAT` is non-zero), the endflags will be posted. But rather than setting the complete variable to true and thus never receiving any further run consideration, the behavior is reset to its initial state. Presumably the user sets endflags that will cause the condition flags to be not immediately satisfied, thus putting the behavior in a state waiting again for an external event flag to be posted. The default value of this parameter is *false*.

8.10 BHV_MemoryTurnLimit

The objective of the Memory-Turn-Limit behavior is to avoid vehicle turns that may cross back on its own path and risk damage to the towed equipment. Its configuration is determined by the two parameters described below which combine to set a vehicle turn radius limit. However, it is not strictly described by a limited turn radius; it stores a time-stamped history of recent recorded headings and maintains a *heading average*, and forms its objective function on a range deviation from that average. This behavior merely expresses a preference for a particular heading. If other behaviors also have a heading preference, coordination/compromise will take place through the multi-objective optimization process. The following parameters are defined for this behavior:

`MEMORY_TIME`: The duration of time for which the heading history is maintained and heading average calculated. The default value is -1, indicating that the parameter is un-set. In this case the behavior will not produce an objective function.

`TURN_RANGE`: The range of heading values deviating from the current heading average outside of which the behavior reflects sharp penalty in its objective function. The default value is -1, indicating that the parameter is un-set. In this case the behavior will not produce an objective function.

The heading history is maintained locally in the behavior by storing the currently observed heading and keeping a queue of n recent headings within the `MEMORY_TIME` threshold. The heading average calculation below handles the issue of angle wrap in a set of n headings $h_0 \ldots h_{n-1}$ where each heading is in the range $[0, 359]$.

$$\text{heading\_avg} = \text{atan2}(s, c) \cdot 180/\pi,$$

where s and c are given by:

$$s = \sum_{k=0}^{n-1} \sin\left(h_k \pi/180)\right), \qquad c = \sum_{k=0}^{n-1} \cos\left(h_k \pi/180)\right).$$

The vehicle turn radius r is not explicitly a parameter of the behavior, but is given by:

$$r = v/((u/180)\pi),$$

where v is the vehicle speed and u is the turn rate given by:

$$u = \texttt{TURN\_RANGE}/\texttt{MEMORY\_TIME}.$$

The same turn radius is possible with different pairs of values for `TURN_RANGE` and `MEMORY_TIME`. However,m larger values of `TURN_RANGE` allow sharper initial turns but temper the turn rate after the initial sharper turn has been achieved.

A Rendering of the MemoryTurnLimit Objective Function

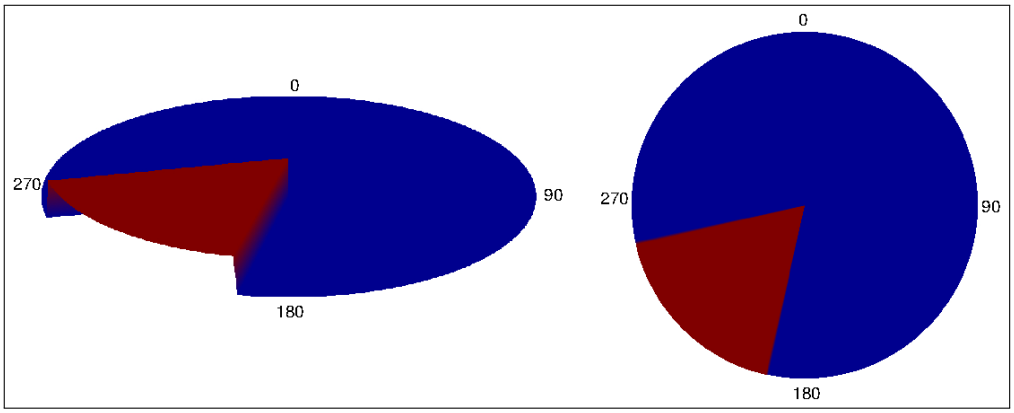


Figure 34: **The MemoryTurnLimit objective function**: The objective function produced by the MemoryTurnLimit behavior is defined over possible heading values. Depicted here is an objective function formed when the recent heading history is 225 degrees and the `turn_range` parameter is set to 30 degrees. The resulting objective function highly favors headings in the range of 190-240 degrees. One the right is a "birds-eye" view of the function, and on the right the function is viewed at an angle to appreciate the 3D quality of the function. Higher (red) values correspond to higher utility.

8.11 BHV_StationKeep

8.11.1 Overview

This behavior is designed to keep the vehicle at a given lat/lon or x,y station-keep position by varying the speed to the station–point as a linear function of its distance to the point. The parameters allow one to choose the two distances between which the speed varies linearly, the range of linear speeds, and a default transit speed if the vehicle is outside the outer radius.

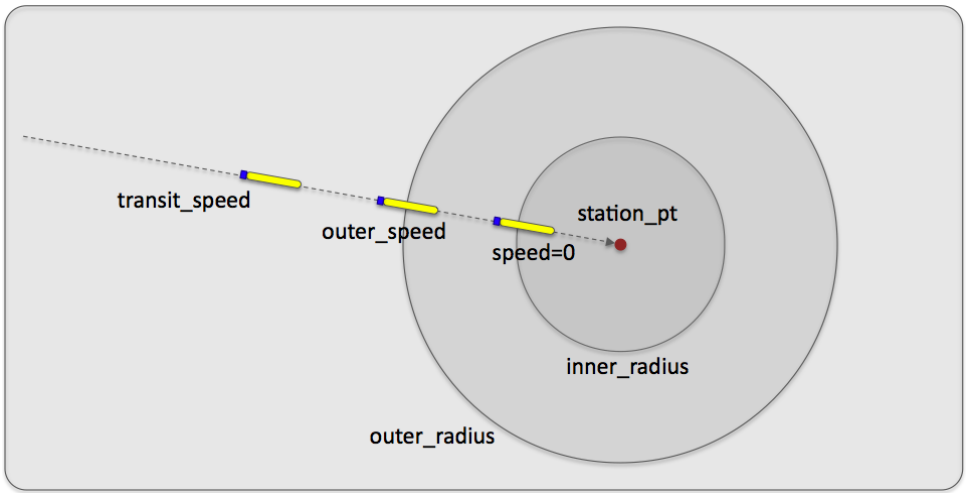


Figure 35: **The station-keep behavior parameters:** The station-keep behavior can be configured to approach the outer station circle with a given transit speed, and will decrease its preference for speed linearly between the outer radius and inner radius. The preferred speed is zero when the vehicle is at or inside the inner radius.

An alternative to this station keeping behavior is an active loiter around a very tight polygon with the `BHV_LOITER` behavior. This station keeping behavior conserves energy and aims to minimize propulsor use. The behavior can be configured to station-keep at a pre-set point, or wherever the vehicle happens to be when the behavior transitions into an active state.

The station-keep behavior was initially developed for use on an autonomous kayak. It's worth pointing out that a vehicle's control system, i.e., the front-seat driver described in Section 2.3, may have a native station-keeping mode, in which case the activation of this behavior would be replaced by a message from the backseat autonomy system to invoke the station-keeping mode. It's also worth pointing out that most UUVs are positively buoyant and will simply come to the surface if commanded with a zero-speed.

8.11.2 Brief Summary of the BHV_StationKeep Behavior Parameters

The following parameters are defined for this behavior. A more detailed description is provided other parts of this section, and in Table 41 on page 276.

Parameter	Description BHV_StationKeep
STATION_PT	An x,y pair given as a point in local coordinates.
POINT	A supported alias for STATION_POINT.
CENTER_ACTIVE	If true, station-keep at position upon activation.
INNER_RADIUS	Distance to station-point within which the preferred speed is zero.
OUTER_RADIUS	Distance within which the preferred speed begins to decrease.
OUTER_SPEED	Preferred speed at outer radius, decreasing toward inner radius.
SWING_TIME	Duration of drift of station circle with vehicle upon activation.
TRANSIT_SPEED	Preferred speed beyond the outer radius.
EXTRA_SPEED	A deprecated alias for TRANSIT_SPEED.
HIBERNATION_RADIUS	A radius used for low-power, passive station-keeping.

8.11.3 Setting the Station-Keep Point and Radial-Speed Relationships

The station-keep point is set in one of two ways: either with a pre-specified fixed position, or with the vehicle's current position when the vehicle transitions into the running state. To set a fixed station-keep position:

```
station_pt = 100,250
```

To configure the behavior to station-keep at the vehicle's current position when it enters the running state:

```
center_active = true  // "true" is case insensitive
```

At the outset of station-keeping via center_activate, the vehicle typically is moving at some speed. Despite the fact that station-keeping is immediately active and typically results in a desired speed of zero if no other behaviors are active, the vehicle will continue some distance before coming to a near or complete stop in the water, thus "over-shooting" the station-keep point. This often means that the station-keep behavior will immediately turn the vehicle around to come back to the station-keep point. This can be countered by setting the behavior's "swing time" parameter, the amount of time after initial center-activation that the station-keep point is allowed to drift with the current position of the vehicle before becoming fixed. The format is:

```
swing_time = <time-duration>   // default is 0
```

The <time-duration> is given in seconds and the duration is clipped by the range $[0, 60]$.

If the behavior enters the running state, but center-activation is not set to true, and no pre-specified fixed position is given, the behavior will not produce an objective function. It will remain in the running state, but not the active state. (See Section 6.5.3 for more detail on behavior run states.) In this situation, a warning will be posted: BHV_WARNING="STATION_POINT_NOT_SET".

The INNER_RADIUS and OUTER_RADIUS parameters affect the preferred speed of the behavior as it relates to the vehicle's current range to the station point. The preferred speed at the outer radius is given by the parameter OUTER_SPEED. The preferred speed decreases linearly to zero as the vehicle approaches the inner radius. The default values for the inner and outer radii are 4 and 15

respectively. If configured with values such that the inner is greater then the outer, this will not trigger an error, but the two radii parameters will be collapsed to the value of the inner radius on the first iteration of the behavior.

8.11.4 Passive Low-Energy Station Keeping Mode

The station-keep behavior can be configured to operate in a "passive" mode. This mode differs from the default mode primarily in the way it acts after it reaches the inner-radius, i.e., the point at which the behavior regards the vehicle to be on-station and outputs a preferred speed of zero. In the normal mode, the behavior will begin to output a preferred heading and non-zero speed as soon as the vehicle slips beyond the inner-radius. In the passive mode, the behavior will let the vehicle drift or otherwise move to a distance specified by the `HIBERNATION_RADIUS` before it resumes outputting a preferred heading and non-zero speed. The idea is shown in Figure 36.

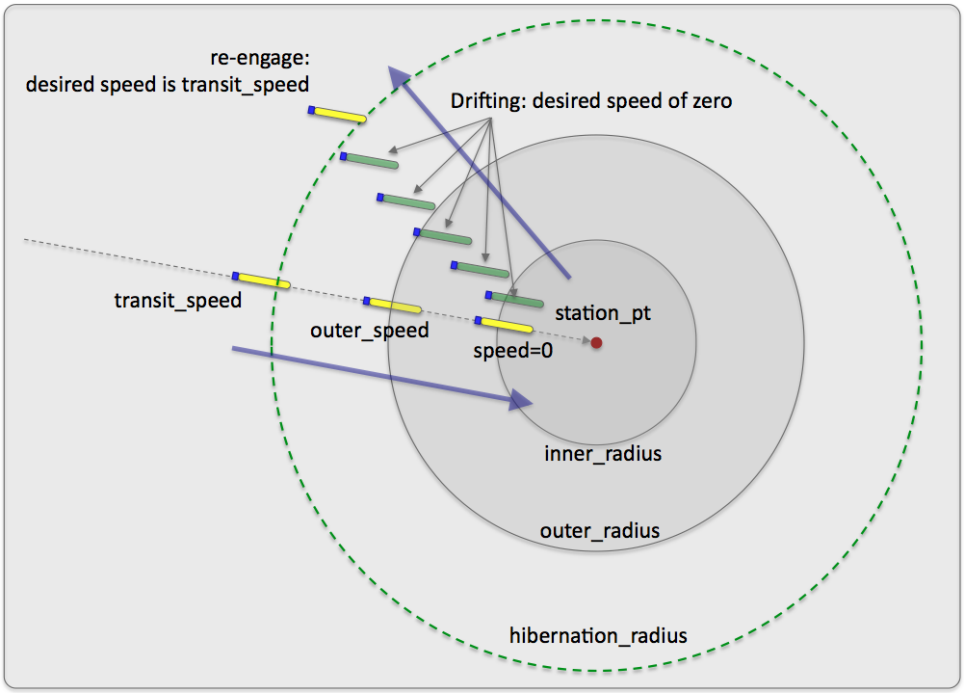


Figure 36: **Passive station-keeping:** The station-keep behavior can be configured in the "passive" mode. The vehicle will move toward the station point until it reaches the `inner_radius` or until progress ceases. It will then drift until its distance to the station point is beyond the `hibernation_radius`. At this point it will re-engage to reach the station-point and may trigger another behavior to dive.

This mode was built with UUVs in mind. Most UUVs are deployed having a positive buoyancy (battery dies - vehicle floats to the surface). They need to be moving at some speed to maintain a depth. Furthermore, it may not be safe to assume that a UUV can effectively execute a desired heading when it is operating on the surface. For these reasons, when operating in the passive mode, this behavior will publish a variable indicating whether it is in the mode of drifting or attempting to make progress toward the station point. The status is published in the variable `PSKEEP_MODE`,

short for "passive station-keeping mode". This variable will be set to `"SEEKING_STATION"` when outputting a non-zero speed preference, and presumably moving toward the station-point. The variable will be set to `"HIBERNATING"` otherwise. This opens the option of configuring the helm with the ConstantDepth behavior to work in conjunction with the StationKeep behavior by conditioning the ConstantDepth behavior to be running only when `PSKEEP_MODE="SEEKING_STATION"`. The idea is shown in Figure 37.

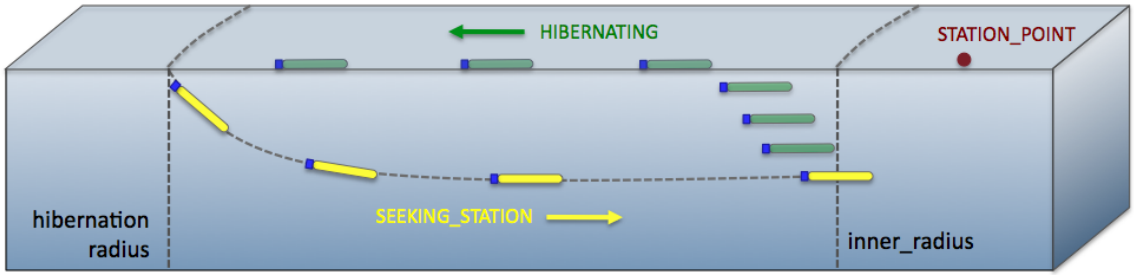


Figure 37: **Passive station-keeping with depth coordination:** The passive mode can be coordinated with the ConstantDepth behavior to dive each time the StationKeep behavior enters the `"SEEKING_STATION"` mode. This ensures that a UUV needing to be at depth to have reliable heading control will indeed be at depth when it needs to be.

This behavior mode is regarded as "low-power" due to the presumably long periods of drifting before resuming actively seeking the station point. A couple of safeguards are designed to ensure that when the behavior is in the `"STATION_SEEKING"` mode, that it does not get hung or stuck in this mode for much longer than intended or needed. How could one become stuck in this mode? Two ways - by either reaching an equilibrium at-speed, (and perhaps at-depth) state where the vehicle is neither progressing toward or way from the `inner_radius`, or by repeatedly "missing" the `inner_radius` by heading right past it.

Both cases can be guarded against and detected by monitoring the history of vehicle speed in the direction of the station-point. If this speed becomes zero, an equilibrium state is assumed, and if it becomes negative, it is assumed that the vehicle missed the inner radius circle entirely. In short, the StationKeep behavior exits the `"STATION_SEEKING"` mode and enters the `"HIBERNATING"` mode when it detects the vehicle speed toward the station-point reach zero. To calculate this vehicle speed, a ten-second history of range to the station-point is kept by the behavior. A zero speed, or "stale-progress" criteria is declared simply if the range to the station-point for the most recent measure in the history is not less than the range of ten seconds ago in the history list. The behavior will transition into the `"HIBERNATING"` mode if either the inner-radius or stale-progress criteria are met.

It is also possible that when the StationKeep behavior enters the `"SEEKING_STATION"` mode from the `"HIBERNATING"` mode, that the vehicle initially begins to open its range to the station-point before it begins to close range. This would be expected, for example, if the vehicle were pointed away from the station-point when the behavior first entered the `"SEEKING_STATION"` mode. In this case it's quite possible that the behavior would correctly, but unwantingly, infer that the stale-progress criteria has been met. For this reason, the stale-progress criteria is not applied until an

"initial-progress" criteria is met after entering the "SEEKING_STATION" mode. The same ten second history is used to detect when the vehicle begins to make initial progress, i.e., closing range, toward the station-point.

8.11.5 Station Keeping On Demand

A common, and perhaps recommended configuration, is to have one station-keep behavior defined for a given helm configuration and have it set to be usable in one of three ways: (a) station-keep at a default pre-specified position, (b) station-keep at a specified position dynamically provided, or (c) station-keep at the vehicle's present position when activated. The behavior would be configured as follows:

```
STATION_PT    = 100,200 // The default station-keep point
CENTER_ACTIVE = false
UPDATES       = STATION_UPDATES
CONDITION     = STATION_REQUEST = true
```

Then, to use the station-keep behavior in the above three ways, the following three pairs of postings, i.e., pokes, to the MOOSDB would be used. See Section 7.2.2 for more on the UPDATES parameter defined for all behaviors - by utilizing this dynamic configuration hook, the one behavior configuration above can be used in these different manners. The first pair would result in the behavior keeping station at its pre-arranged point of 100,200:

```
STATION_REQUEST = true
STATION_UPDATES = CENTER_ACTIVATE=false"
```

The second line above dynamically configures the behavior parameter CENTER_ACTIVATE to be false to ensure that the point given by the original STATION_PT parameter is used. Even though the CENTER_ACTIVATE parameter is initially set to false, the above usage sets it to false anyway, to be safe, and in case it has been dynamically set to true in a prior usage.

In the second case below, again the CENTER_ACTIVATE parameter is dynamically set to false for the same reasons. In this case the STATION_POINT parameter is also dynamically configured with a given point:

```
STATION_REQUEST = true
STATION_UPDATES = "STATION_PT=45,-150 # CENTER_ACTIVATE=false"
```

In the last case, below, the behavior is activated and configured to station-keep at the vehicle's present position when activated. There is no need to tinker with the STATION_PT parameter since this parameter is ignored when CENTER_ACTIVATE is true:

```
STATION_REQUEST = true
STATION_UPDATES = "CENTER_ACTIVATE=true"
```

It's worth noting that above variable-value pairs that trigger the station-keep behavior could have come from a variety of sources. They could be endflags from another behavior. They could have come from a poke using uPokeDB, uTermCommand, pMarineViewer or any third party command and control interface.

8.12 BHV_Timer

The Timer behavior is a somewhat unique behavior in that it never produces an objective function. It has virtually no functionality beyond what is derived from the parent IvPBehavior class. It can be used to set a timer between the observation of one or more events (with condition flags) and the posting of one or more events (with end flags). The `DURATION`, `DURATION_STATUS`, `CONDITION`, `RUNFLAG` and `ENDFLAG` parameters are all defined generally for behaviors. There are no additional parameters defined for this behavior.

8.12.1 Brief Overview of Configuration Parameters and Variables Published

This behavior publishes two variables for monitoring and logging performance - `TIMER_IDLE`, `TIMER_RUNNING`.

8.13 BHV_TestFailure

The TestFailure behavior is used to test the helm in two conceivable behavior failure modes. First, it may be used to simulate a behavior that crashes and thereby results in the crash of the helm. Second, it may be used to simulate a behavior that consumes a sufficiently large enough amount of time so as cause the helm to be considered "hung" by consumers of the helm output.

It may be worth recalling that the helm is compiled, with behaviors, into a single MOOS application. Although some behaviors may be compiled into shared libraries loaded at run time, thereby not requiring a recompile, all behaviors do run as part of a single helm process. *A crashed behavior results in a crashed helm.* Furthermore the helm, on each iteration, queries each participating behavior for its input. It does not do this in separate threads, and there is no timeout with a default reply should a behavior never answer. *A hung behavior results in a hung helm.* These are architecture decisions that on one hand allow a substantial amount of simplicity in the helm implementation and debugging. Furthermore, it's not clear that a graceful and safe policy exists to safely handle a rogue behavior other than to either (a) abort the mission or (b) put the vehicle in the hands of a much more conservatively configured "standby" instance of the helm, perhaps just to get the vehicle home. This behavior is used to simulate both kinds of rogue behaviors, a behavior that crashes and a behavior the hangs. The crash is implemented simply with an `assert(0)` statement, and the hang is implemented with long for-loop.

8.13.1 Configuration Parameters and Variables Published

Beyond the configuration parameters defined for all behaviors, described in Section 7.2, there is only one additional configuration parameter for this behavior

Parameter	Description BHV_TestFailure
FAILURE_TYPE:	Fail by `crash`, or fail by `hang`.

Table 14: Configuration parameters for the BHV_TestFailure behavior.

Choosing to fail with a *crash* will result in behavior executing an `assert(0)` as soon as the behavior enters the running state. Configuring for a *hang* will likewise result in the execution of a long for-loop upon entering the running state. The hang time is by default three seconds, but may be altered by specifying a time alongside the hang parameter as in line 9 below.

Below are a couple typical usage configurations. In the first case, the TestFailure behavior is set with a condition as on line 4, to remain idle until the vehicle is deployed. The duration parameter on line 6, defined for all helm behaviors, ensures the behavior will not run until two minutes into the mission. This is done presumably to have the failure occur while the vehicle is "in the middle of doing something". Setting the `duration_idle_decay` parameter to false indicates the duration countdown is not to proceed while the behavior is still idle. In this case the failure type is set to hang for two seconds. This behavior configuration is used in the Kilo example mission, described in Section XYZ, and can be run to illustrate.

```
1  Behavior = BHV_TestFailure
2  {
3    name      = test_failure
4    condition = DEPLOY=true
```

```
 5
 6    duration  = 120
 7    duration_idle_decay = false
 8
 9    failure_type = hang,2
10  }
```

Another configuration style below will result in the behavior crashing as soon as the behavior meets its run condition. Presumably the user can simply poke the MOOSDB at any time to invoke the failure, or another MOOS app may generate the poke at the key desired moment. The `failure_type` is left unspecified since `crash` is the default failure type.

```
 1  Behavior = BHV_TestFailure
 2  {
 3    name      = test_failure
 4    condition = MUD_HITS_THE_FAN = true
 5  }
```

9 Contact Related Behaviors of the IvP Helm

The section contains is a description of four behaviors currently written for the IvP Helm that reason about relative position to another vehicle or *contact*.

- The AvoidCollision behavior
- The CutRange behavior
- The Shadow behavior
- The Trail behavior

Each behavior needs to know about the position and trajectory of a given contact. The helm subscribes to the MOOS variable, `NODE_REPORT`, which may have content similar to:

```
NODE_REPORT= "NAME=alpha,TYPE=UUV,MOOSDB_TIME=39.01,UTC_TIME=1252348077.59,
              X=51.71,Y=-35.50,LAT=43.824981,LON=-70.329755,SPD=2.00,HDG=118.85,
              YAW=118.84754,DEPTH=4.63,LENGTH=3.8,MODE=SURVEYING"
```

This contact information is stored in the helm information buffer for any behavior that wishes to retrieve it on any iteration.

Contact related behaviors may be configured to operate on a fixed particular contact, or they may be configured as templates where the contact name is provided dynamically at run time as new instances of the behavior are spawned. Configuring behaviors enabled for dynamic spawning is discussed in Section 7.7 on page 93. Contact related behaviors are not spawned automatically for new contacts. A separate contact manager process, `pBasicContactMgr`, manages the node reports and posts alerts based on the contact range. The alerts are MOOS variable-value pairs that may be coordinated with the behavior configuration to spawn new behaviors. The `pBasicContactManager` application is discussed in Section 15, and the Berta mission in Section 10.6 shows a working example of the contact manager used with the helm to dynamically spawn collision avoidance behaviors.

9.1 Properties Common to All Contact Related Behaviors

Contact related behaviors are distinct from non contact related behaviors in that they share a fair amount of functionality in dealing with their contact of interest. Contact related behaviors are implemented as a subclass of the `IvPContactBehavior` class, which is a subclass of the `IvPBehavior` class. Much of the shared functionality of contact related behaviors is implemented in the former. The shared functionality includes several common configuration parameters, and mechanisms for reasoning about the closest point of approach (CPA) between the platform and contact for candidate maneuver considerations. These topics are discussed next, prior to the sections on the behaviors themselves.

9.1.1 Common Behavior Configuration Parameters

The following set of parameters are common to all the contact related behaviors:

Parameter	Description
CONTACT	Name or unique identifier of a contact to be avoided.
DECAY	Time interval during which extrapolated position slows to a halt.
EXTRAPOLATE	If "true", contact position is extrapolated from last position and trajectory.
ON_NO_CONTACT_OK	If false, a helm error is posted if no contact information exists.
TIME_ON_LEG	The time on leg, in seconds, used for calculating closest point of approach.

Table 15: Configuration parameters common to contact-related behaviors.

The CONTACT parameter specifies the contact name or identifier. This name is used as a key by the behaviors for querying the contact position and trajectory. It must match the contact name received by the helm in an incoming NODE_REPORT message. The contact name may be specified at helm launch time, but it may also be specified at run time if the behavior is configured as a template for dynamic spawning. The latter is more common, for example, in a collision avoidance behavior where the name or ID of the contact is not known until a contact manager alerts the helm. See the Berta mission in Section 10.6 for an example of this usage.

The EXTRAPOLATE and DECAY paramters are used to address the situation where a contact/node report has significant delays between updates. Extrapolation is enabled by setting the EXTRAPOLATE parameter to true, which is the default. The behavior may be configured to have limited extrapolation by setting a decay time interval. The extrapolated position is based on the last known contact position, heading and speed. The speed used for calculations may begin decaying at the beginning of the decay interval and will have decayed to zero at the end of the decay interval. The default setting is "decay = 15, 30", in seconds. The idea is shown in Figure 38.

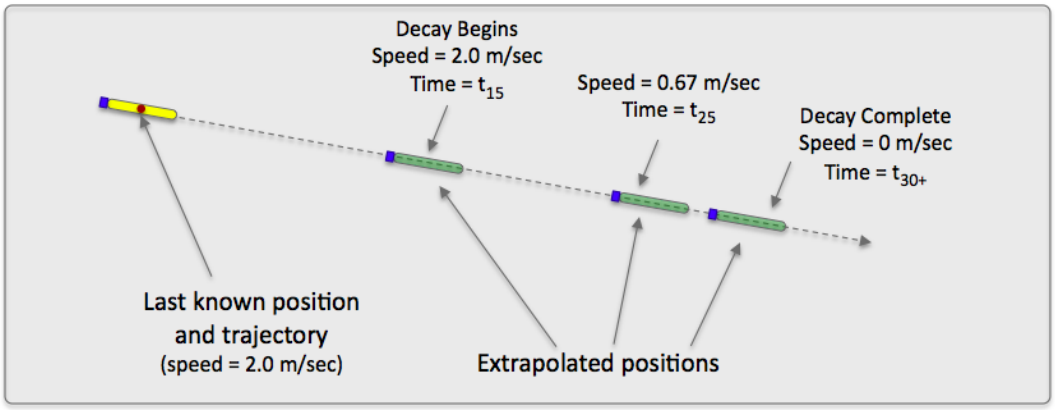


Figure 38: **Contact Extrapolations:** Contact related behaviors may use an extrapolated position of the contact to compensate for periods of no new reports for the contact. A decay period may be used to effectively halt the extrapolated contact position after some specified period of time. In the example in this figure, the decay window is $[15, 30]$ seconds. After 30 seconds, the extrapolated position does not extend further.

The ON_NO_CONTACT_OK parameter determines how the behavior should regard the situation where it is unable to find any information about a given contact. If this parameter is set to true, the default, then the behavior will post a warning, BHV_WARNING if no contact information is found. Otherwise the behavior will post an error with BHV_ERROR. In the latter case the helm may interpret this as request to halt the helm and come to zero speed and depth.

The TIME_ON_LEG parameter refers to the behavior's calculations of the closest point of approach (CPA) for candidate maneuver legs. A candidate maneuver leg is defined by a the heading, speed, and time-on-leg components. Longer time-on-leg settings tend to report deceivingly worrisome CPA distances even for contacts at a great distance, and lower time-on-leg settings tend to report deceivingly comfortable CPA distances even for vehicles at relatively low distances. The default setting for this parameter is 60 seconds.

9.1.2 Closest Point of Approach Calculations

The IvP functions produced by contact-related behaviors are defined over the domain of posssible heading and speed choices. The utility assigned to a point in this domain (a heading-speed pair) depends in part on the calculated closest point of approach (CPA) between the candidate maneuver leg, and the contact leg formed from the contact's position and trajectory. Figure 39 shows the relationship $cpa(\theta, v)$ between CPA and candidate maneuvers (θ, v), where θ=heading and v=speed, for a given relative position between ownship and a given contact vehicle and trajectory. The IvP function generated by the AvoidCollision behavior applies a further user-defined utility function to the CPA calculation for a candidate maneuver, $f(cpa(\theta, v))$. The form of $f()$ is determined by configuration parameters specific to the individual behavior.

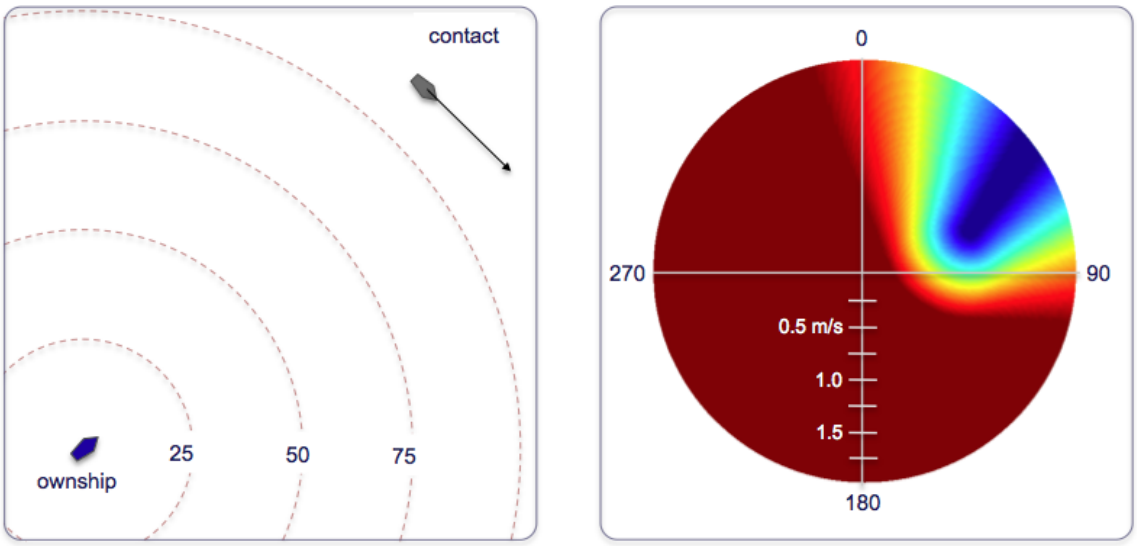


Figure 39: **The closest point of approach mapping:** The function on the right indicates the relative change in calculated closest point of approach between ownship and contact position and trajectory shown on the left.

For contact related behaviors, an important quality of a candidate action $\langle \theta, v, t \rangle$, is the *closest point of approach (CPA)* between two vehicles during a candidate leg. A behavior producing an

objective function with CPA as a component of its utility function needs to perform many variations of this calculation on each new call to generate an IvP objective function. The algorithm is given here, highlighting a few areas where caching may be exploited to improve efficiency.

Our own current position is known and given by $(x,\ y)$, and the other vehicle's current position and trajectory is given by $(x_b,\ y_b,\ \theta_b,\ v_b)$. To compute the CPA distance for a given $\langle \theta, v, t \rangle$, first the time t_{min} when the minimum distance between two vehicles occurs is computed. The distance between the two vehicles at the current time can by determined by the Pythagorean theorem. Generally, for any given time t (where the current time is $t = 0$), and assuming the other vehicle stays on a constant trajectory, the distance between the two vehicles for any chosen $\langle \theta, v, t \rangle$ is given by:

$$dist^2(\theta, v, t) = k_2 t^2 + k_1 t + k_0, \tag{1}$$

where

$$\begin{aligned}
k_2 = \ & \cos^2(\theta)v^2 - 2\cos(\theta)v\cos(\theta_b)v_b + \cos^2(\theta_b)v_b^2 + \sin^2(\theta)v^2 - \\
& 2\sin(\theta)v\sin(\theta_b)v_b + \sin^2(\theta_b)v_b^2 \\
k_1 = \ & 2\cos(\theta)vy - 2\cos(\theta)vy_b - 2y\cos(\theta_b)v_b + 2\cos(\theta_b)v_b y_b + \\
& 2\sin(\theta)vx - 2\sin(\theta)vx_b - 2x\sin(\theta_b)v_b + 2\sin(\theta_b)v_b x_b \\
k_0 = \ & y^2 - 2yy_b + y_b^2 + x^2 - 2xx_b + x_b^2
\end{aligned}$$

The stationary point is obtained by taking the first derivative with respect to t:

$$dist^2(\theta, v, t)' = 2k_2 t + k_1.$$

Since there is no "maximum" distance, this stationary point always represents the closest point of approach, and therefore:

$$t' = \frac{-k_1}{2k_2}.$$

The value of t_{min} may be in the past, i.e., less than zero, if the two vehicles are currently opening range. Or t_{min} may be well beyond t, the time length of the candidate maneuver $\langle \theta, v, t \rangle$. Therefore the value of t_{min} is clipped by $[0, t]$. Furthermore t_{min} is zero when the two vehicles have the same heading and speed (the only condition where k_2 is zero). The actual CPA value is then obtained by plugging t_{min} back into (1).

$$CPA(\theta, v, t) = \sqrt{k_2 {t_{min}}^2 + k_1 t_{min} + k_0}. \tag{2}$$

As mentioned before, this calculation is a common component in the underlying utility function for behaviors dealing with relative vehicle motion. A behavior, within a single iteration of the control cycle, will perform a sequence of calculations on different $\langle \theta, v, t \rangle$ values. However, all calculations have the same values of current vehicle position $(x,\ y)$, and current position and trajectory of the other vehicle $(x_b,\ y_b,\ \theta_b,\ v_b)$. To make this overall sequence of calculations faster,

all terms in (1) comprised exclusively of x, y, x_b, y_b, θ_b, v_b are calculated once and cached for later calculations.

9.2 The AvoidCollision Behavior

The AvoidCollision behavior will produce IvP objective functions designed to avoid collisions (and near collisions) with another specified vehicle. The IvP functions produced by this behavior are defined over the domain of posssible heading and speed choices. The utility assigned to a point in this domain (a heading-speed pair) depends in part on the calculated closest point of approach (CPA) between the candidate maneuver leg, and the contact leg formed from the contact's position and trajectory. A further user-defined utility function is applied to the CPA calculation for a candidate maneuver.

9.2.1 Brief Overview of Configuration Parameters and Variables Published

The configuration parameters and variables published collectively define the interface for the behavior. A more detailed description of usage is provided other parts of this section.

Configuration Parameters for the BHV_AvoidCollision Behavior

The following are the parameters for this behavior, in addition to the configuration parameters defined for all behaviors, described in Section 7.2 beginning on page 81.

Parameter	Description BHV_AvoidCollision
BEARING_LINES	Color code specification for rendering bearing lines, off by default.
COMPLETED_DIST	Range to contact outside of which the behavior completes and dies.
MAX_UTIL_CPA_DIST	Range to contact outside which a considered maneuver will have max utility.
MIN_UTIL_CPA_DIST	Range to contact within which a considered maneuver will have min utility.
PWT_GRADE	Grade of priority growth as the contact moves from the PWT_OUTER_DIST to the PWT_INNER_DIST. Choices are linear, quadratic, or quasi.
PWT_INNER_DIST	Range to contact within which the behavior has maximum priority weight.
PWT_OUTER_DIST	Range to contact outside which the behavior has zero priority weight.
CONTACT	Name or unique identifier of a contact to be avoided.
DECAY	Time interval during which extrapolated position slows to a halt.
EXTRAPOLATE	If "true", contact position is extrapolated from last position and trajectory.
ON_NO_CONTACT_OK	If false, a helm error is posted if no contact information exists.
TIME_ON_LEG	The time on leg, in seconds, used for calculating closest point of approach.

Table 16: Configuration parameters for the BHV_AvoidCollision behavior.

Variables Published by the BHV_Waypoint Behavior

The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be supressed or changed to a different variable name using the post_mapping configuration parameter described in Section 7.2.1 on page 82.

Variable	Description BHV_AvoidCollision
CLOSING_SPD_AVD	The current closing speed, in meters per second, to the contact.
CONTACT_RESOLVED	Posted with contact name when the behavior completes and dies.
RANGE_AVD	The current range, in meters, to the contact.
VIEW_SEGLIST	A bearing line between ownship and the contact if configured for rendering.

Table 17: Variables posted by the BHV_AvoidCollision behavior.

Configuring and Using the AvoidCollision Behavior

The AvoidCollision behavior produces an objective function based on the relative positions and trajectories between the vehicle and a given contact. The objective function is based on applying a utility to the calculated closest point of approach (CPA) for a candidate maneuver. The user may configure a priority weight, but this weight is typically degraded as a function of the range to the contact. The behavior may be configured for avoidance with respect to a known contact, or it may be configured to spawn a new instance upon demand as contacts present themselves.

Specifying the Priority Policy - the pwt_*_dist Parameters

The AvoidCollision behavior may be configured to increase its priority as its range to the contact diminishes. The priority weight specified in its configuration represents the *maximum* possible priority applied to the behavior, presumably in close range to the contact. The range at which this maximum priority applies is specified in the PWT_INNER_DIST parameter. Likewise, the PWT_OUTER_DIST parameter specifies a range to the contact where the priority weight becomes zero, regardless of the priority weight specified in the configuration file. This relationship is shown in Figure 40.

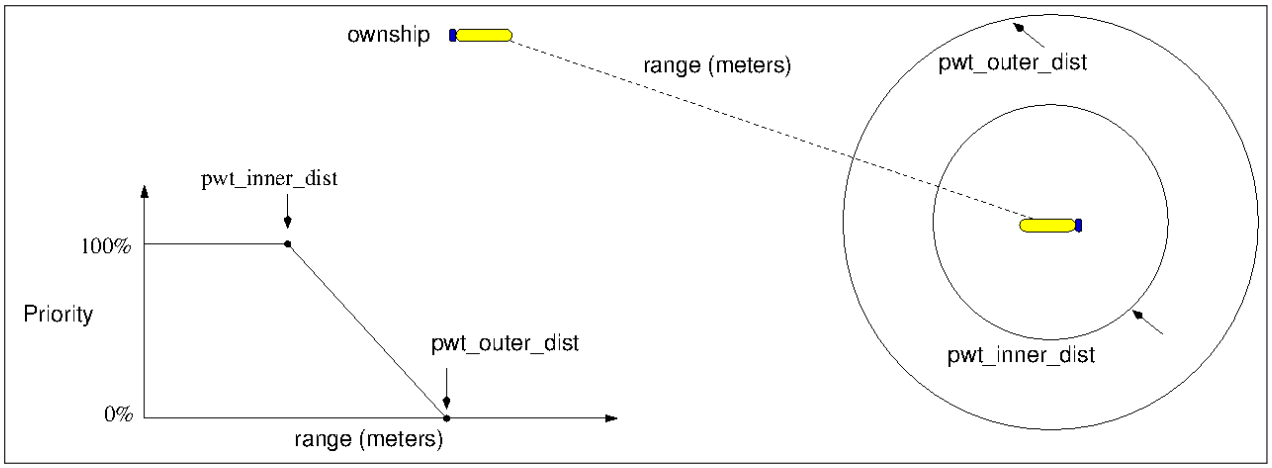


Figure 40: **Parameters for the BHV_AvoidCollision behavior:** The *ownship* vehicle is the platform running the helm. The range between the two vehicles affects whether the behavior is active and with what priority weight. Beyond the range specified by PWT_OUTER_DIST, the behavior is not be active. Within the range of PWT_INNER_DIST, the behavior is active with 100% of its configured priority weight.

By default, the priority weight decreases linearly between the two depicted ranges. The `PWT_GRADE` parameter allows the degradation from maximum priority to zero priority to fall more steeply by setting `PWT_GRADE=quadratric`.

Associating Utility to CPA of Candidate Maneuvers

`MIN_UTIL_CPA_DIST:` The distance (in meters) between ownship and the contact at the closest point of approach (CPA) for a candidate maneuver, below which the behavior treats the distance as it would an actual collision between the two vehicles.

`MIN_UTIL_CPA_DIST:` The distance (in meters) between ownship and the contact at the closest point of approach (CPA) for a candidate maneuver, above which the behavior treats the distance as having the maximum utility.

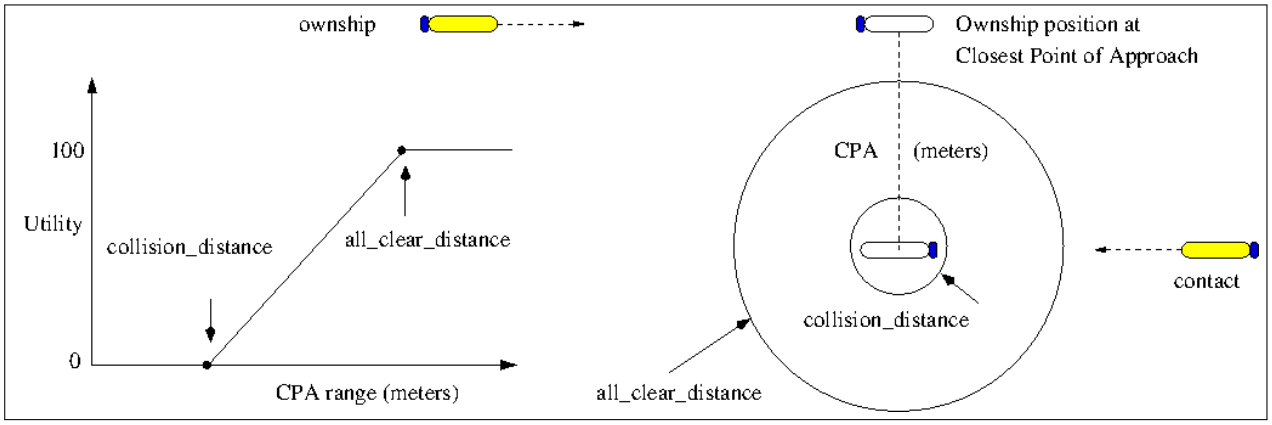


Figure 41: Parameters for the BHV_AvoidCollision behavior. The *ownship* vehicle is the platform running the helm. The `collision_distance` is used when applying a utility metric to a calculated closest point of approach (CPA) for a candidate maneuver. A CPA less than or equal to the `collision_distance` is treated as an actual collision with the lowest utility rating.

9.3 BHV_CutRange

This behavior will drive the vehicle to reduce the range between itself and another specified vehicle (nearly the opposite of the BHV_AvoidCollision behavior). The following parameters are defined for this behavior:

9.3.1 Brief Overview of Configuration Parameters and Variables Published

The behavior configuration parameters, and variables published by the behavior, collectively define the interface for the behavior. A more detailed description of usage is provided other parts of this section.

Configuration Parameters for the BHV_Waypoint Behavior

The following are the configuration parameters for this behavior, in addition to the configuration parameters defined for all behaviors, described in Section 7.2 beginning on page 81.

Parameter	Description BHV_CutRange
PWT_OUTER_DIST	Range to contact outside which the behavior has maximum priority.
PWT_INNER_DIST	Range to contact within which the behavior has zero priority.
GIVEUP_DIST	Range to contact outside which the behavior will give up (become inactive).
PATIENCE	Linear scale choice between preferring heading directly to the contact (patience=0) or heading on the lowest closest point of approach (patience=100).
CONTACT	Name or unique identifier of a contact to be avoided.
DECAY	Time interval during which extrapolated position slows to a halt.
EXTRAPOLATE	If "true", contact position is extrapolated from last position and trajectory.
ON_NO_CONTACT_OK	If false, a helm error is posted if no contact information exists.
TIME_ON_LEG	The time on leg, in seconds, used for calculating closest point of approach.

Table 18: Configuration paramters for the BHV_CutRange behavior.

Variables Published by the BHV_CutRange Behavior

The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be supressed or changed to a different variable name using the post_mapping configuration parameter described in Section 7.2.1 on page 82.

Variable	Description BHV_CutRange
VIEW_SEGLIST	A bearing line between ownship and the contact if configured for rendering.

Table 19: Variables posted by the BHV_CutRange behavior.

9.3.2 Specifying the Priority Policy - the pwt_*_dist Parameters

DIST_PRIORITY_INTERVAL: Two distance values given by a comma-separated pair min,max where the min value is the range at or below which the behavior will have a zero priority. The min value is the range at or above which the behavior will have 100% of its statically assigned priority. The percentage between the two values scales linearly.

TIME_ON_LEG: The behavior uses a closest-point-of-approach (CPA) calculation to evaluate candidate heading-speed maneuvers. The CPA calculation is based on a 60 second maneuver by default, but this time duration can be altered with this parameter.

GIVE_UP_RANGE: The range between ownship and the contact at or above which the behavior will cease to provide output (the objective function) to influence the vehicle heading and speed. By default this value is zero which is interpreted as infinity - it will never give up.

PATIENCE: The PATIENCE parameter ranges between 0 and 100 and is clipped automatically if out of range. A value of 0 will result in the behavior attempting to steer the vehicle directly toward the current position of the contact. A value of 100 will result in an attempt to steer toward the closest point of approach given the current linear track of the contact, and the prevailing setting of the TIME_ON_LEG parameter.

9.4 BHV_Shadow

This behavior will drive the vehicle to match the trajectory of another specified vehicle. This behavior in conjunction with the BHV_CutRange behavior can produce a "track and trail" capability. The following parameters are defined for this behavior:

Parameter	Description BHV_Shadow
PWT_OUTER_DIST	Range to contact outside which the behavior has zero priority.
HEADING_PEAKWIDTH	Width of the peak, in degrees, in the produced ZAIC-style IvP function.
HEADING_BASEWIDTH	Width of the base, in degrees, in the produced ZAIC-style IvP function.
SPEED_PEAKWIDTH	Width of the peak, in m/sec, in the produced ZAIC-style function.
SPEED_PEAKWIDTH	Width of the base, in m/sec, in the produced ZAIC-style IvP function.
CONTACT	Name or unique identifier of a contact to be avoided.
DECAY	Time interval during which extrapolated position slows to a halt.
EXTRAPOLATE	If "true", contact position is extrapolated from last position and trajectory.
ON_NO_CONTACT_OK	If false, a helm error is posted if no contact information exists.

Table 20: Configuration paramters for the BHV_Shadow behavior.

MAX_RANGE: The distance (in meters) that the contact must be within for the behavior to be active and produce an objective function. The default is max_range value is zero meaning it will be active regardless of the distance to the contact.

HEADING_PEAKWIDTH: This behavior uses the ZAIC_PEAK tool from the IvP Toolbox for generating an objective function over heading and speed. This parameter sets the peakwidth parameter of the heading component.

HEADING_BASEWIDTH: This behavior uses the ZAIC_PEAK tool from the IvP Toolbox for generating an objective function over heading and speed. This parameter sets the basewidth parameter of the heading component.

SPEED_PEAKWIDTH: This behavior uses the ZAIC_PEAK tool from the IvP Toolbox for generating an objective function over heading and speed. This parameter sets the peakwidth parameter of the speed component.

SPEED_BASEWIDTH: This behavior uses the ZAIC_PEAK tool from the IvP Toolbox for generating an objective function over heading and speed. This parameter sets the basewidth parameter of the speed component.

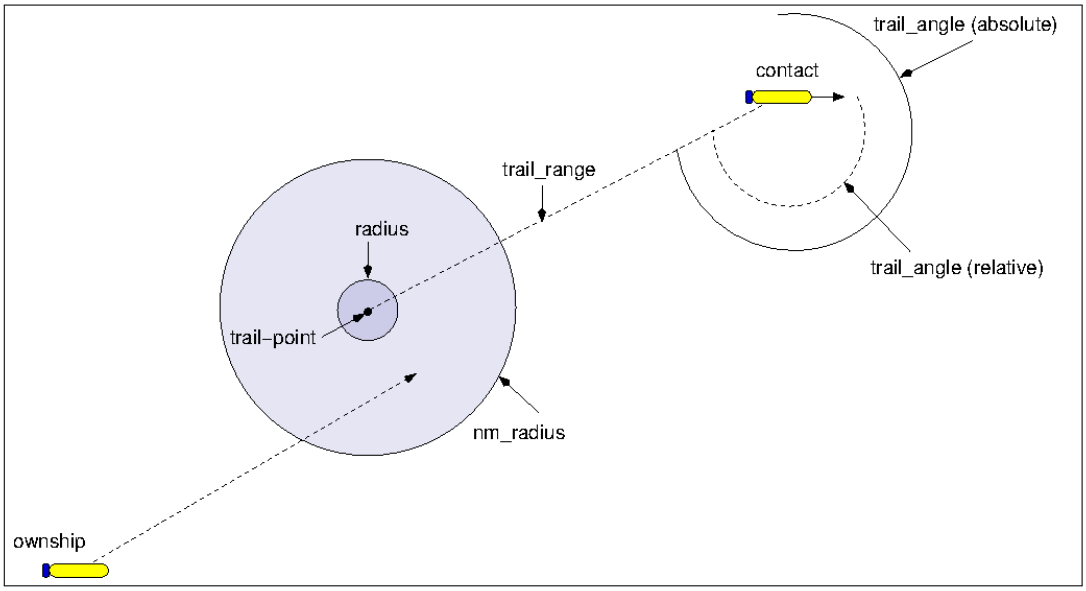


Figure 42: Interpolation of vehicle speed inside the radius set by `NM_RADIUS` relative to the extrapolated trail position.

.

9.5 BHV_Trail

This behavior will drive the vehicle to trail or follow another specified vehicle at a given relative position. A tool for "formation flying". The following parameters are defined for this behavior:

9.5.1 Brief Overview of Configuration Parameters and Variables Published

The behavior configuration parameters, and variables published by the behavior, collectively define the interface for the behavior. A more detailed description of usage is provided other parts of this section.

Configuration Parameters for the BHV_Trail Behavior

The following are the configuration parameters for this behavior, in addition to the configuration parameters defined for all behaviors, described in Section 7.2 beginning on page 81.

Parameter	Description BHV_Trail
PWT_OUTER_DIST	Range to contact outside which the behavior has zero priority.
NM_RADIUS	If in this range to contact and ahead of it, slow down.
RADIUS	If outside this radius to the contact, head to nm_radius ahead of trail point
TRAIL_ANGLE	Relative angle to the contact to set the trail-point.
TRAIL_ANGLE_TYPE	Either "relative" or "absolute" bearing/angle to the contact.
TRAIL_RANGE	Relative distance to the contact to set the trail-point.
CONTACT	Name or unique identifier of a contact to be avoided.
DECAY	Time interval during which extrapolated position slows to a halt.
EXTRAPOLATE	If "true", contact position is extrapolated from last position and trajectory.
ON_NO_CONTACT_OK	If false, a helm error is posted if no contact information exists.
TIME_ON_LEG	The time on leg, in seconds, used for calculating closest point of approach.

Table 21: Configuration paramters for the BHV_Trail behavior.

Variables Published by the BHV_CutRange Behavior

The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be supressed or changed to a different variable name using the post_mapping configuration parameter described in Section 7.2.1 on page 82.

Variable	Description BHV_CutRange
VIEW_SEGLIST	A bearing line between ownship and the contact if configured for rendering.

Table 22: Variables posted by the BHV_Trail behavior.

9.5.2 Specifying the Priority Policy - the pwt_*_dist Parameters

TRAIL_RANGE: The range component of the relative position to the contact to trail.

TRAIL_ANGLE: The relative angle of the relative position to the contact to trail. (180 is directly behind, 90 is a parallel track to the contacts starboard side, -90 is on the port side of the contact.)

TRAIL_ANGLE_TYPE: The trail angle may be set to either relative (the default), or absolute.

RADIUS: The distance (in meters) from the trail position that will result in the behavior "cutting range" to the trail position, and inside of which will result in the behavior "shadowing" the contact. The default is 5 meters.

NM_RADIUS: The distance in meters from the trail point within which the speed will be gradually change from the outer chase speed (max speed) and the speed of the contact, as illustrated in Fig. 42. This parameter should typically be set to several times the value of RADIUS to achieve smooth formation flying. Default is 20 meters.

`MAX_RANGE:` The distance (in meters) that the contact must be within for the behavior to be active and produce an objective function. The default is max_range value is zero meaning it will be active regardless of the distance to the contact.

10 Extended Example Missions with the IvP Helm

This section addresses a number of example mission configurations the user may run in simulation, to explore various aspects of the helm, certain behaviors, and certain tools for configuring, running and analyzing missions. All missions are available in the moos-ivp software tree in the following location:

```
moos-ivp/ivp/missions/
```

Obtaining the moos-ivp software tree from the public server was discussed in Section 1.4.

10.1 Preliminaries

Each mission in the `missions/` directory is contained in a dedicated subdirectory such as `s1_alpha`, or `m2_berta`. The `'s'` and `'m'` in the prefix indicate whether it is a single-vehicle or multi-vehicle mission. The numbering roughly correlates to an increase in complexity from commonly used bread-and-butter behaviors and configurations to more complex mission configurations.

Each directory contains one or more *mission* files, i.e., files that end in the `.moos` suffix, and one or more *behavior* files, i.e., files that end in the `.bhv` suffix. During the course of running a mission, certain additional temporary files may be created. In all cases, the temporary files may be removed by running the `clean.sh` script included in each directory. The configuration and launching of the missions requires some familiarity with three tools, (a) pAntler, for launching multiple MOOS applications, (b) the `nsplug` tool, for building composite `.moos` and `.bhv` files from a template and set of include files, and (c) basics of shell scripts for organizing the overall launch process and cleanup. A brief recap of these three tools is covered next.

10.1.1 Using pAntler to Launch Missions

The `pAntler` utility is a command-line tool used to orchestrate the launching of a group of MOOS processes. A typical invocation is:

```
> pAntler charlie.moos
```

The MOOS file contains a configuration block for each process, and an Antler configuration block which is typically the first block in the file. Each block contains a collection of local "parameter = value" pairs read in by each individual application. There are also typically a number of global parameter=value pairs for possible use by all applications.

10.1.2 Using the nsplug Utility for Configuring Mission and Behavior Files

In multi-vehicle missions, the maintenance of mission and behavior files across multiple vehicles can become cumbersome and user error-prone. Since many of the file components are identical between vehicles, a fair amount of the problem is mitigated by building the files from file template files that include the common components from separate files. The `nsplug` command-line utility is used to expand a template file into a target file, with arguments passed that typically distinguish the produced file from others produced from the same template. For example, two mission files, one for the vehicle "charlie", and one for the vehicle "frankie" may differ only in their vehicle names and IP Port number, and may be produced from a template file with the following two commands:

```
> nsplug  meta_vehicle.moos targ_charlie.moos VNAME=charlie PORT=9201
> nsplug  meta_vehicle.moos targ_frankie.moos VNAME=charlie PORT=9202
```

For the multi-vehicle example missions distributed with the moos-ivp tree and described in this section, this use of templates is used in all cases. The mission directories downloaded will not contain the `nsplug` "output" files such as `targ_charlie.moos` above. Rather, they are built by executing the launch scripts provided in each directory with a command such as:

```
> ./launch.sh
```

The above invocation will both build the mission and behavior files and invoke pAntler for each vehicle to start the simulation. Certain common command-line options such as `"./launch --just_build"`, to build the mission files without launching, are available in all scripts. A few conventions with respect to templates and `nsplug` are used across all example mission directories.

1. Files used as templates have a `"meta_"` prefix.
2. Files meant to be included or plugged into another template file have a `"plug_"` prefix.
3. Output files that are built from templates and plug-ins via `nsplug` have `"targ_"` prefix.
4. Template or plug-in files that contribute to a target MOOS file have `".moos"` suffix.
5. Template or plug-in files that contribute to a target Helm behavior file have `".bhv"` suffix.

For example, in the `m2_berta` mission, the following files exist prior to building any target files and launching:

```
> cd missions/m2_berta
> ls
clean.sh*                  plug_uSimMarine.moos        plug_pMOOSBridgeV.moos
launch.sh*                 plug_origin_warp.moos       plug_pMarinePID.moos
meta_shoreside.moos        plug_pBasicContactMgr.moos  plug_pNodeReporter.moos
meta_vehicle.bhv           plug_pHelmIvP.moos          plug_uProcessWatch.moos
meta_vehicle.moos          plug_pLogger.moos           plug_uXMS.moos
```

After building all the target files, the following additional `targ_*` files exist:

```
> ./launch.sh --just_build
> ls
clean.sh*                  plug_pBasicContactMgr.moos  plug_uXMS.moos
launch.sh*                 plug_pHelmIvP.moos          targ_gilda.bhv
meta_shoreside.moos        plug_pLogger.moos           targ_gilda.moos
meta_vehicle.bhv           plug_pMOOSBridgeV.moos      targ_henry.bhv
meta_vehicle.moos          plug_pMarinePID.moos        targ_henry.moos
plug_uSimMarine.moos       plug_pNodeReporter.moos     targ_shoreside.moos
plug_origin_warp.moos      plug_uProcessWatch.moos
```

The mission may launched, with `MOOSTimeWarp=10`, by:

```
> ./launch.sh 10
```

10.2 Mission S3: The Charlie Mission

Overview of Charlie Mission Components and Topics

Mission:	"charlie" in moos-ivp/missions/s3_charlie
Behaviors:	BHV_Loiter, BHV_StationKeep, BHV_Waypoint
MOOS Apps:	pHelmIvP, pLogger, uSimMarine, pMarinePID, pNodeReporter, pMarineViewer, uTimerScript
Primary Topics:	(1) Hierarchical Mode Declarations, page 152 (2) The Loiter behavior, page 154 (2a) Loiter traversal - Clockwise vs. counter-clockwise, page 154 (2b) Loiter polygon shapes, hexagons, ellipses, page 155 (2c) Loiter dynamic changes to the polygon center position, page 155 (2d) Loiter robustness to periodic external forces, page 155 (3) The StationKeep Behavior, page 156
Side Topics:	(1) uTimerScript is used to simulate wind gusts, external forces (2) Use of pMarineViewer action buttons and action pull-down menu

Launching the Charlie Mission

The charlie mission may be launched from the command line in the following manner:

```
> cd moos-ivp/missions/s3_charlie/
> ./launch.sh --warp=10
```

This should bring up a pMarineViewer window like that shown in Figure 44 on page 158, with a single vehicle, "henry", initially in the PARK mode. After hitting the DEPLOY button in the lower right corner, the vehicle enters the "LOITERING" mode and begins to proceed to the polygon shown. To get a quick feel for what's possible in this simulation, the following are some things to try. (a) Click on the "CWISE" and "CCWISE" buttons to change the direction of loitering around the polygon. (b) Click the RETURN and DEPLOY buttons repeatedly to see the vehicle switch between two primary modes. (c) Select the "WIND_GUSTS=true" option in the ACTION pull-down menu to see the vehicle adjust to random periodic external forces. (d) Select the "STATION_KEEP=true" option in the ACTION pull-down menu to see how the vehicle station keeps with simulated wind gust. (e) Let the vehicle return to its return point and watch how it automatically switches to the station keeping mode, and then re-deploy it. (f) Select the ellipse polygon from the ACTION pull-down menu to see a different loiter shape. (g) Select the different center points for the loiter ellipse from the ACTION pull-down menu to see just the location of the loiter polygon change and the vehicle adjust.

10.2.1 Topic #1: Hierarchical Mode Declarations in the Charlie Mission

The Charlie mission is organized by hierarchical mode declarations into four modes: INACTIVE, LOITERING, RETURNING, and STATION-KEEPING. The mode declarations are given a the top of the charlie.bhv file and shown again in Figure 43. The pMarineViewer application by default shows the vehicle's mode alongside the vehicle name, and when the simulation is first launched, the Charlie vehicle is shown on the screen with the mode PARK.

```
// Excerpt from charlie.bhv

set MODE = ACTIVE {
  DEPLOY = true
} INACTIVE

set MODE = LOITERING {
  MODE = ACTIVE
  LOITER = true
  RETURN != true
  STATION_KEEP != true
}

set MODE = RETURNING {
  MODE = ACTIVE
  (RETURN = true)
  STATION_KEEP != true
} STATION-KEEPING
```

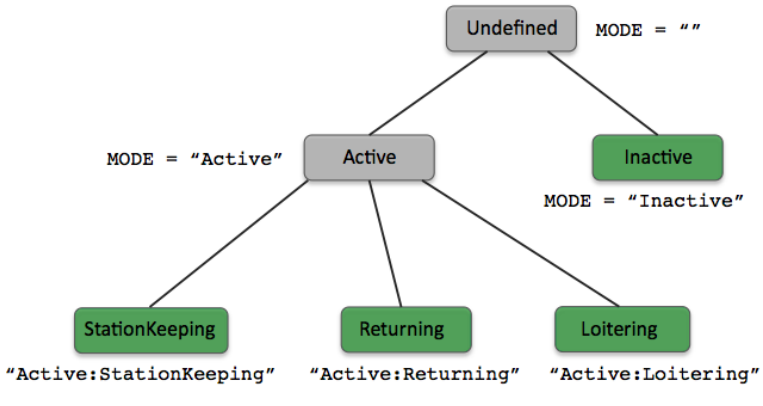


Figure 43: **Charlie Mission Mode Declarations:** The helm, when in drive, will be in one of the four modes indicated by the leaf nodes on the tree. The hierarchical mode structure means, for example, that when the vehicle is in the `"RETURNING"` mode, it is also considered to be in the `"ACTIVE"` mode. The vehicle mode, by default, is displayed alongside the vehicle name in the pMarineViewer window.

The `"PARK"` mode displayed with the vehicle when the simulation is launched is actually the high level helm state. As described in Section 5.2, the helm has a higher level *helm state* which is either `PARK` or `DRIVE`, analogous to a car in either the "Park" or "Drive" modes. When the Charlie mission is first launched, the helm is not yet in drive, and is put into drive when the `DEPLOY` button is hit. This button is configured in the `pMarineViewer` configuration block to make several MOOS pokes with a single button click. It posts `MOOS_MANUAL_OVERRIDE=false`, to put the vehicle in the `DRIVE` helm state. It posts `DEPLOY=true` to put the helm in the `ACTIVE` mode, and it posts `LOITER=true` to put the helm in the `ACTIVE:LOITERING` mode.

The Charlie mission is configured to allow for easy transition between the other modes via the pMarineViewer interface. The vehicle may be put into the `STATION-KEEPING` mode at any time via the `STATION_KEEP=true` option in the `ACTION` pull-down menu in pMarineViewer. This does nothing more than make the `STATION_KEEP=true` post to the MOOSDB. The same could have been accomplished by uPokeDB for example:

```
> cd moos-ivp/missions/s3_charlie/
> uPokeDB charlie.moos STATION_KEEP=true
```

Referring to the mode declarations in Figure 43, the `LOITERING` and `RETURNING` modes both have the condition `STATION_KEEP != true`, so both of those mode requirements fail to be satisfied. The `STATION-KEEPING` mode is the default mode, i.e., the "else" mode when the requirements of the `RETURNING` mode are not met. The `STATION_KEEP=true` posting could also have been made by another MOOS process, for example, connected to an acoustic modem, an Iridium satellite, or wifi interface.

The helm mode is communicated to pMarineViewer from the helm via the MOOS variable `IVPHELM_SUMMARY`. This variable is parsed and the mode is rendered next to the vehicle name in the viewer. This rendering may be turned off by toggling through the "name options", with the `'n'` key, or setting the `vehicles_name_viewable` parameter in the pMarineViewer configuration block to one of the values list in Table **??** on page **??**.

10.2.2 Topic #2: The Loiter Behavior

The LOITERING mode in the Charlie mission is characterized by the BHV_Loiter behavior, and we explore some of capabilities and options for this behavior here. The behavior configuration, in file charlie.bhv is shown in Listing 17 below. The first configuration block, in lines 3-6, are for parameters defined generally for all IvP Helm behaviors, and the second block, in lines 8-16, are for parameters defined explicitly for the Loiter behavior.

Listing 17 - Configuration of the Loiter behavior in the Charlie mission, from the file charlie.bhv.

```
 0  //---------------------------------------------
 1  Behavior = BHV_Loiter
 2  {
 3    name      = loiter
 4    priority  = 100
 5    condition = MODE==LOITERING
 6    updates   = UP_LOITER
 7
 8            speed = 1.3
 9        clockwise = false
10    capture_radius = 4.0
11        nm_radius = 15.0
12          polygon = format=radial, x=0, y=-75, radius=40, pts=6, snap=1,
13      visual_hints = nextpt_color=white, nextpt_lcolor=khaki
14      visual_hints = edge_color=blue, vertex_color=yellow
15      visual_hints = edge_size=1, vertex_size=2, label=LOITER_POLYGON
16  }
```

When the vehicle is first deployed, it heads to the hexagon shaped polygon shown in Figure 44, configured in line 12. It traverses the polygon in a counter-clockwise manner due to the configuration in line 9. The Loiter behavior maintains an internal mode, the "acquire_mode" which is true when the vehicle is not sufficiently close to the polygon, defined by the acquire_dist parameter which is by default 10 meters. It also keeps track of how many vertex arrivals come by way of achieving the capture radius, and how many by way of the non-monotonic radius. These internal state variables are summarized by the Loiter behavior by publishing a variable LOITER_REPORT. An example might look like:

```
LOITER_REPORT = "index=2,capture_hits=12,nonmono_hits=3,acquire_mode=false"
```

In the Charlie mission, the pMarineViewer is configured to scope on this variable by default, and the evolving LOITER_REPORT may be monitored as the vehicle progresses around the polygon.

Loiter Traversal - Clockwise vs. Counter-Clockwise

The traversal direction of the Loiter behavior is by default clockwise and can be set to counter-clockwise with the parameter clockwise=false. In the Charlie mission, the traversal direction can be changed by selecting UP_LOITER=clockwise=true and UP_LOITER=clockwise=false from the ACTION pull-down menu. This is a good way to observe the algorithm used by the behavior to acquire the polygon when sufficiently far inside the polygon. The traversal direction could also have been affected by any other MOOS application capable of executing the above two MOOS pokes, including uPokeDB, or any application connected to a communications device.

Loiter Polygon Shapes, Hexagons, and Ellipses

The only restriction on the shape traversed by the Loiter behavior is that it be a convex polygon. The typical specification of the polygon is shown in line 12, via the "Radial" syntax, which accepts a center position (`"x=0, y=-75"`), a distance from the center point to each vertex (`"radius=40"`), the number of points in the polygon, (`"pts=6"`), and a snap value, which rounds the vertex positions to, in this case, the nearest whole meter (`"snap=1"`). Ellipse shapes are supported with a string such as:

```
polygon = format=ellipse, x=110, y=-75, degs=150, pts=14, major=80, minor=20
```

The above ellipse specification is selectable in the Charlie mission via the `ACTION` pull-down menu. If selected while in the loiter mode, the vehicle immediately begins traversing to this new polygon, as shown in Figure 45.

Loiter Dynamic Changes to the Polygon Position

In addition to the an outright change to the loiter polygon as described above, which includes shape changes, number of vertices and position, there are a few methods for just changing the polygon position while leaving the other characteristics in tact. The Loiter behavior accepts a parameter for just the center position. The following two such assignments are selectable in the `ACTION` pull-down menu in the pMarineViewer for the Charlie mission:

```
center_assign = 40,-40
center_assign = x=100, y=-80
```

Try selecting either of these options, with the polygon set to either the hexagon or ellipse and confirm that only the position changes. One other method for affecting the polygon position is via the `center_activate` parameter, which is `false` by default. When set to `true`, the polygon center is set to be the vehicle's present position when the Loiter behavior enters the running state. In the Charlie mission, try selecting `center_activate=true` from the `ACTION` pull-down menu. If already in the `LOITERING` mode, then nothing happens immediately. If and when the helm exits and returns to the `LOITERING` mode, for example after clicking the `RETURN` button, and then the `DEPLOY` button again, note that the loiter polygon is centered on the vehicle's present position. The `center_active` feature is particularly useful when the Loiter behavior is used more or less as a station keeping behavior and one wants to be able to command the vehicle to loiter at the present position at any given time.

Loiter Robustness to Periodic External Forces

When the Loiter behavior is active and the vehicle is proceeding around the polygon vertices, it is proceeding more or less like the Waypoint behavior (both behaviors share the same WaypointEngine class as a sub-component). Things get interesting when the vehicle is approaching the polygon from the outside or from a distance internally (for example when the `center_activate` parameter set to `true`. To test and demonstrate this robustness, the Charlie mission is configured with the uTimerScript utility to generate periodic random forces to simulate wind gusts, to push the vehicle off the loiter polygon in unpredictable ways. An example is shown in Figure 46 on page 159.

The simulated wind gusts may be activated by selecting the WIND_GUSTS=true option from the pMarineViewer ACTION pull-down menu. The details of the simulated forces can be found in the uTimerScript configuration block in the charlie.moos file. The script works by posting external force vectors to the MOOS variable USM_FORCE_VECTOR_ADD, which is read by the uSimMarine application to alter the prevailing external force vector, which by default has a magnitude of zero. The syntax of the uTimerScript script to implement the forces in this example are described in more detail in Section 14.9.2 on page 251.

10.2.3 Topic #3: The StationKeep Behavior

The STATION-KEEPING mode in the Charlie mission is characterized by the BHV_StationKeep behavior, and we explore some of capabilities and options for this behavior here. The behavior configuration, in file charlie.bhv is shown in Listing 18 below. The first configuration block, in lines 3-5, are for parameters defined generally for all IvP Helm behaviors, and the second block, in lines 7-11, are for parameters defined explicitly for the Loiter behavior.

Listing 18 - Configuration of the StationKeep behavior in the Charlie mission, from the file charlie.bhv.

```
 0  //-------------------------------------------
 1  Behavior = BHV_StationKeep
 2  {
 3    name      = station-keep
 4    priority  = 100
 5    condition = MODE==STATION-KEEPING
 6
 7    center_activate = true
 8       inner_radius = 5
 9       outer_radius = 10
10        outer_speed = 1.0
11         swing_time = 0
12  }
```

Putting the Vehicle into the Station Keeping Mode

The vehicle is put into the station keeping mode by posting STATION_KEEP=true to the MOOSDB. This results in the helm mode, represented by the MOOS variable MODE, being set to station keeping. The hierarchical mode declarations used in the Charlie mission are declared in the top of the charlie.bhv file and were shown in Figure 43. The StationKeep behavior is conditioned on MODE==STATION-KEEPPING, on line 5 above. The StationKeep behavior may be configured to station keep at a specified point in water (with the station_pt parameter), or it may be configured to station keep wherever it happens to be when it enters or re-enters the running state, as it is configured in the Charlie mission by virtue of line 7 above. In the Charlie mission, pMarineViewer is configured with two on-screen buttons (Section **??**). Two of them are used for toggling the STATION_KEEP variable to the MOOSDB.

What is Happening in the Station Keeping Mode

The station keeping behavior (the only running behavior in this mode) attempts to keep the vehicle within a certain distance, given by the inner_radius parameter, to a center point. Inside this radius,

it simply lets the vehicle drift. Outside the radius it sets a heading and speed to drive the vehicle back to the center point. The speed decreases as it approaches the inner circle, and is at its highest speed (set by the `outer_speed` parameter on line 10 above) when its range to the center point is greater than that given by the `outer_radius` distance. It also makes two postings to the `VIEW_POLYGON` object to represent the circles implied by the `inner_radius` and `outer_radius` parameters, as shown in Figure 47.

Suggestions for Tinkering in the Charlie Mission

- Station keeping is trivial if the vehicle doesn't drift. Try turning on the artificial wind gusts available in the Charlie mission, courtesy of the uTimerScript script, available by selecting `WIND_GUSTS=true` from the Action pull-down menu. The vehicle performance is a bit more interesting now in the station keeping mode.

- Try configuring the simulator, `uSimMarine`, to reflect a vehicle that takes much more time to come to a full stop in the water. This can be done by setting `DECELERATION=0.1` in the `uSimMarine` configuration block, or by posting `USM_DECELERATION=0.1` to the MOOSDB once the mission is running. Note that when the vehicle is put into station keeping mode, its station point is set to the point where it is when it enters this mode. Since the vehicle takes so long to slow down, it immediately drifts out of the inner radius and turns around 180 degrees. This is a typical situation seen in the field. This can be countered a bit by setting the `swing_time` parameter in the StationKeep behavior. This parameter is the number of elapsed seconds after the vehicle enters the station keeping mode before the station keeping behavior marks its present position as the point to station keep around. Try setting `swing_time=5` and re-running the above to see the difference.

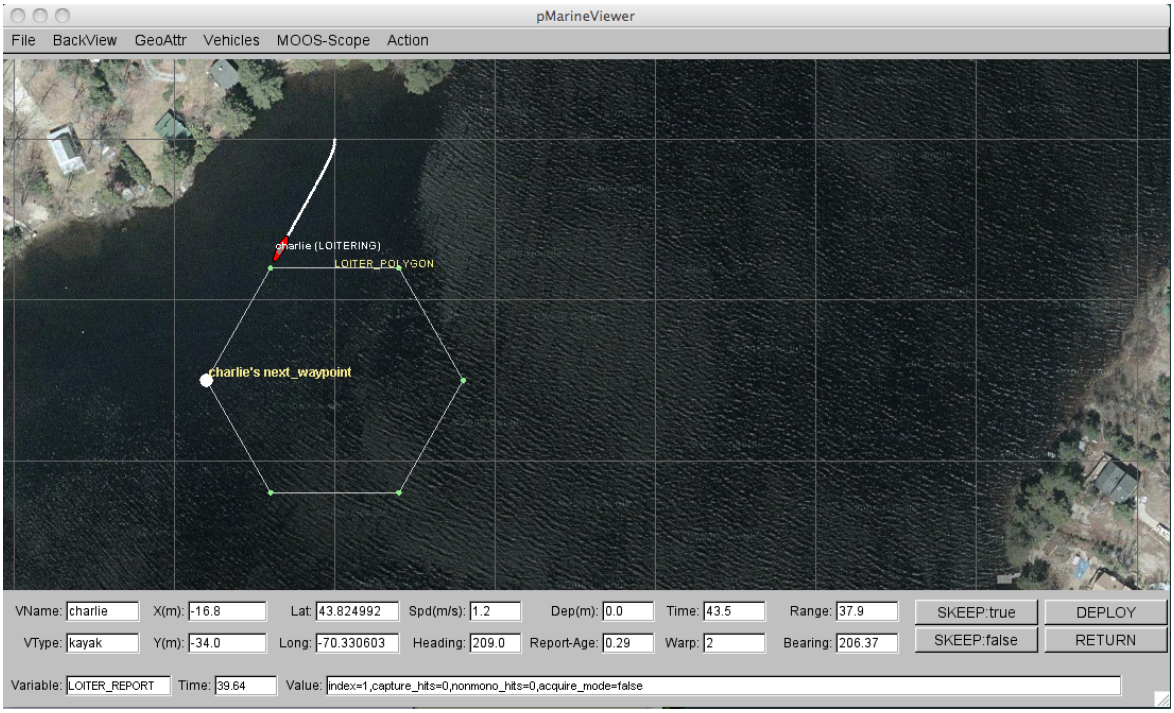


Figure 44: **The Charlie Mission (1):** The vehicle `"charlie"` proceeds to a loiter polygon, traversing in a counter-clockwise manner to the first waypoint labelled `"charlie's next waypoint"`.

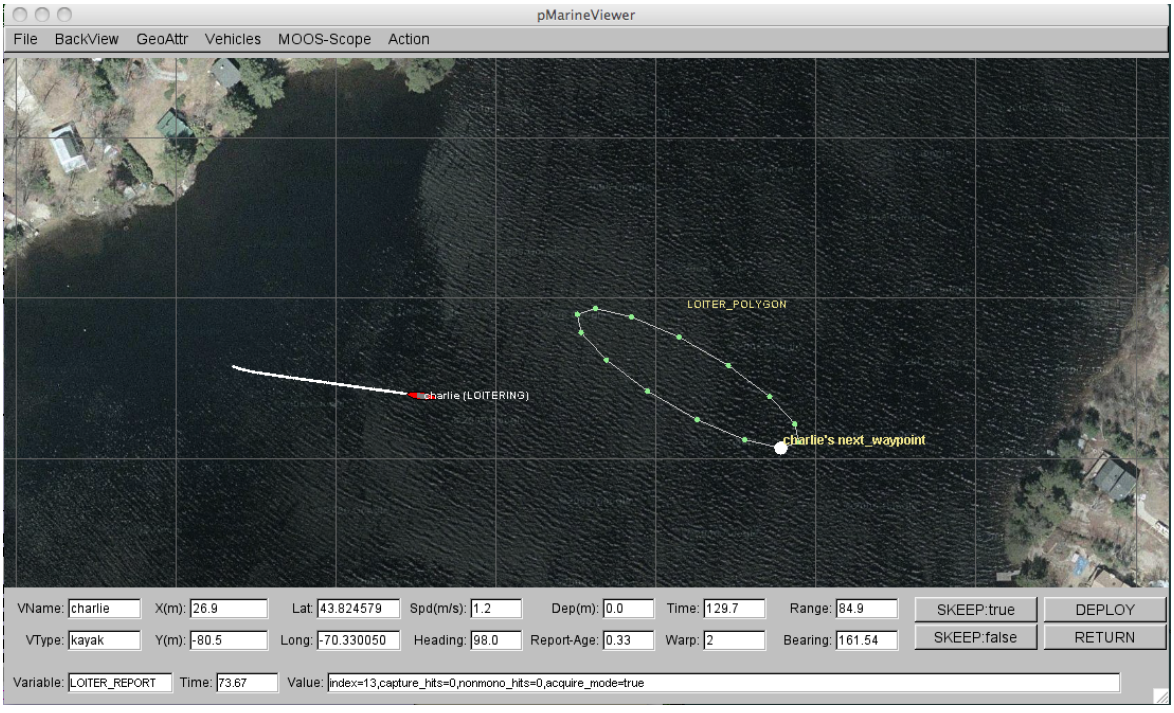


Figure 45: **The Charlie Mission (2):** The loiter traversal polygon for the vehicle `"charlie"` has been dynamically changed to an ellipse via a selection from the `ACTION` pull-down menu.

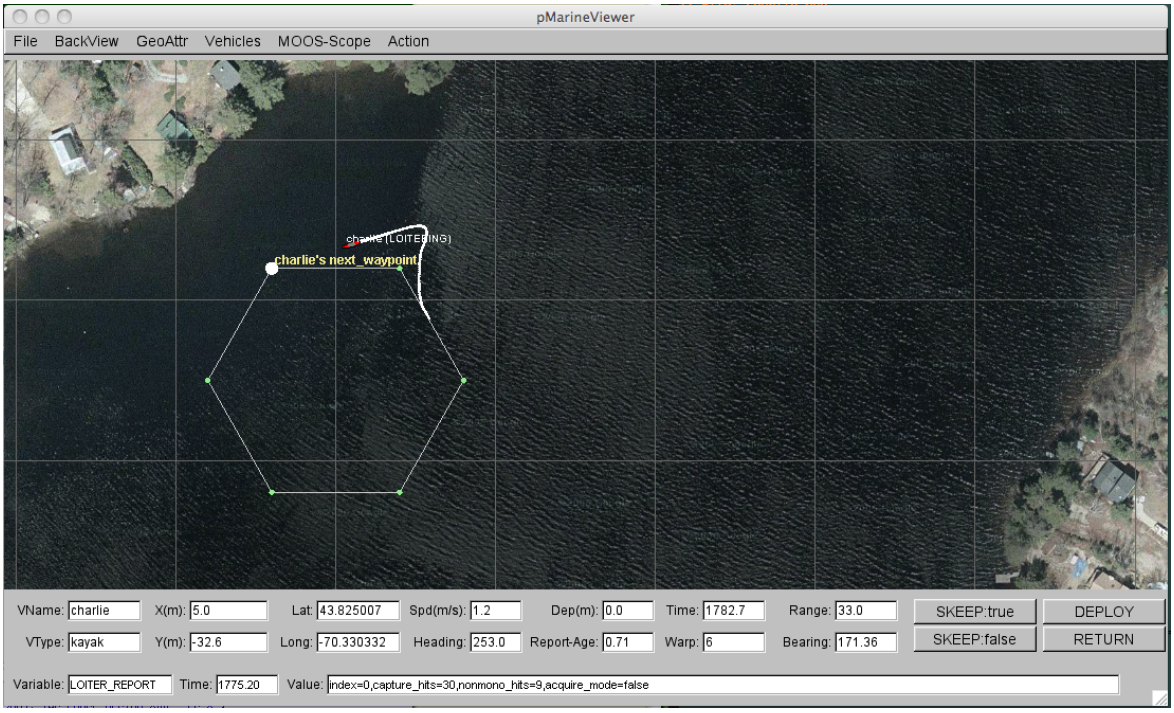


Figure 46: **The Charlie Mission (3):** The vehicle, `"charlie"` recovers from a wind gust and proceeds to re-enter the polygon trajectory at the tangential vertex labelled `"charlie's next waypoint"`.

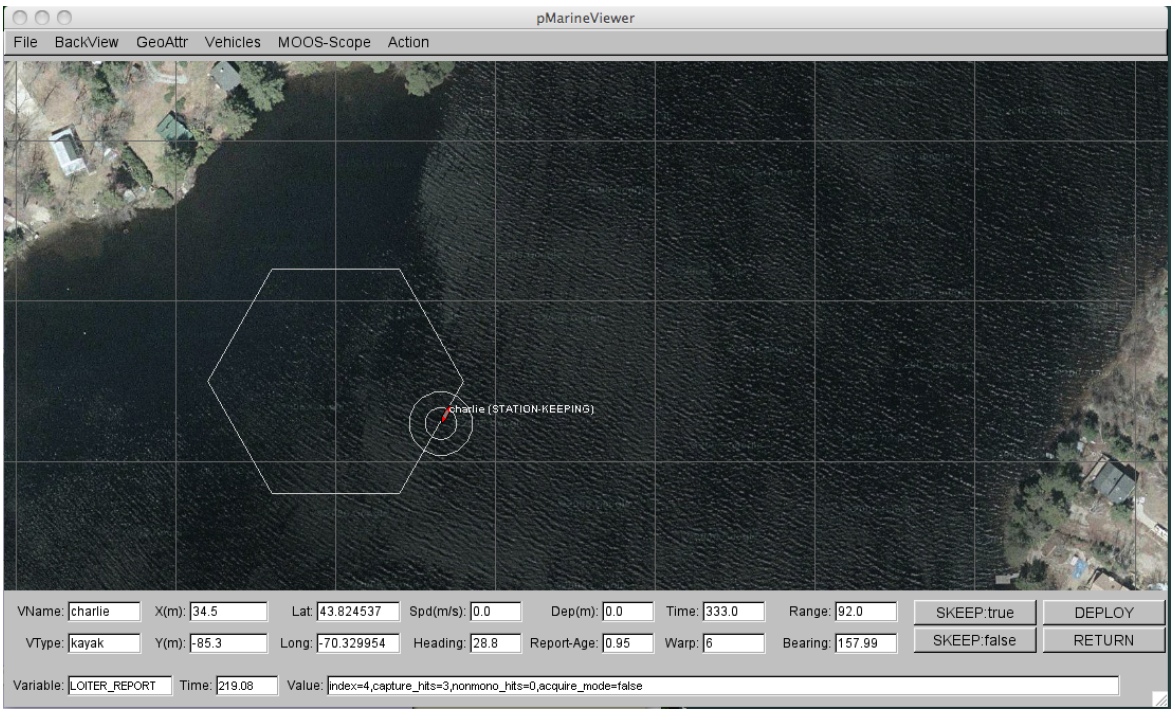


Figure 47: **The Charlie Mission (4):** The vehicle `"charlie"` is put into the station keeping mode. The two rings around the vehicle are the `inner_radius` and `outer_radius` parameters.

10.3 Mission S4: The Delta Mission

The Delta example mission is used to illustrate the operation of a a vehicle in the depth plane (an underwater as opposed to surface vehicle), the illustration of the PeriodicSurface behavior, the use of the Waypoint behavior with survey patterns, and the use of the `pMarineViewer` application to send field-control commands to alter the mission as it unfolds. The vehicle will initially deploy using the Loiter behavior, and will periodically come to the surface. If commanded by the user, the vehicle will break off from the Loiter behavior to execute a survey pattern, after which it will resume loitering at its original location.

Overview of the Delta Mission Components and Topics

Mission:	"delta" in moos-ivp/missions/s4_delta
Behaviors:	BHV_Waypoint, BHV_Loiter, PeriodicSurface, BHV_ConstantDepth
MOOS Apps:	pHelmIvP, pLogger, uSimMarine, pMarinePID, pNodeReporter, pMarineViewer uTimerScript
Primary Topics:	(1) Configuring the Helm for operation at depth (2) The ConstantDepth behavior (3) The PeriodicSurface behavior (4) The Waypoint behavior with survey patterns (5) Using pMarineViewer with Geo-referenced mouse clicks
Side Topics:	(1) uTimerScript is used to simulate a UUV receiving a GPS update.

Launching the Delta Mission

The delta mission may be launched from the command line:

```
> cd moos-ivp/missions/s4_delta/
> ./launch.sh --warp=10
```

This should bring up a `pMarineViewer` window like that shown in Figure 48 on page 166, with a single vehicle, `"dudley"`, initially in the `PARK` helm state. See Section **??** for more on the helm state. After hitting the `DEPLOY` button in the lower right corner, the vehicle enters the `DRIVE` state, and the `LOITERING` helm mode, and begins to proceed along the waypoints as shown. See Section 6.4 for more on more on the helm mode, vs. the helm state. The mode declarations for the Delta mission are defined at the top of the `delta.moos` file, and amount to the following simple hierarchy:

```
o ROOT
|--o INACTIVE
|--o ACTIVE
   |--o RETURNING
   |--o SURVEYING
   |--o LOITERING
```

In the initial helm mode after deployment, the `LOITERING` mode, the helm is running the Loitering behavior, Section 8.3, the ConstantDepth behavior, Section 8.6, and the PeriodicSurface behavior, Section 8.5. It will stay in this mode indefinitely until it is either commanded to return or break off to another region to conduct a survey pattern. In the `LOITERING` mode the vehicle will periodically

come to the surface for a GPS fix, presumably to correct for accumulated navigation error. The `uTimerScript` utility is used to simulate the event of receiving a GPS fix. This is described in more detail in Section 14.9.1.

In the `SURVEYING` mode, the Waypoint behavior becomes active and will execute a survey lawn-mower type pattern, as shown in Figures 49 and 51. In this mode, the periodic surfacing for GPS fixes is suppressed until the pattern is completed. The `SURVEYING` mode is entered whenever the MOOS variable `SURVEY` is set to `true`. In this mission, `pMarineViewer` is configured to toggle the `SURVEY` MOOS variable with on-screen buttons, and it is configured to accept mouse clicks which not only put the vehicle into the `SURVEYING` mode, but also accept the location of the mouse click as the center location of a predefined survey pattern. This kind of `pMarineViewer` configuration is discussed in Section 10.3.5.

10.3.1 Topic #1: Configuring the Helm for Operation at Depth

The Delta mission is configured to simulate a UUV and the helm is configured to produce decisions about the decision variables *heading*, *speed*, as well as *depth*. Adding *depth* to the helm decision space is done in the helm configuration block as shown below in Listing 19, taken from the `delta.moos` mission file. In this case, the *depth* decision space is defined on line 13 with a range from 0 to 500 meters, with a resolution of 1 meter. See Section 5.4 for more on configuring the helm decision space with the `domain` parameter.

Listing 19 - Configuring pHelmIvP in the Delta mission to use behaviors concerning vehicle depth.

```
 1  //--------------------------------------
 2  // pHelmIvP config block
 3
 4  ProcessConfig = pHelmIvP
 5  {
 6    AppTick    = 4
 7    CommsTick  = 4
 8
 9    Behaviors  = delta.bhv
10    Verbose    = quiet
11    Domain     = course:0:359:360
12    Domain     = speed:0:4:21
13    Domain     = depth:0:500:501
14  }
```

In this example, the *depth* decision variable is a mandatory variable, along with *heading* and *speed*. This means that, on any given helm iteration, there *must* be a decision made about depth, or else the helm will post a helm error. How does the helm ensure that a depth decision is always part of any decision? The helm must be configured such that a behavior that reasons about *depth* is always active regardless of the helm mode. In the Delta mission, the three primary modes, `LOITERING`, `SURVEYING`, `RETURNING`, are all sub-modes of the `ACTIVE` mode, as shown above. The ConstantDepth behavior is configured to be running whenever the helm is in the Active mode.

10.3.2 Topic #2: The ConstantDepth Behavior

The ConstantDepth behavior is used in the Delta mission to keep the vehicle a prescribed depth while it is transiting, surveying, and loitering. For simplicity a single ConstantDepth behavior is

used, but a different behavior instance could also be used for each vehicle mode. The behavior configuration, in file `delta.bhv` is shown in Listing 20 below. The first part of the configuration block, in lines 3-6, are for parameters defined generally for all IvP Helm behaviors, and the second part, on line 8, is a parameter defined for the ConstantDepth behavior.

Listing 20 - The ConstantDepth behavior in the Delta mission, from the file delta.bhv.

```
0  //--------------------------------------------
1  Behavior = BHV_ConstantDepth
2  {
3    name      = bhv_const_depth
4    pwt       = 100
5    duration  = no-time-limit
6    condition = MODE==ACTIVE
7
8        depth = 15
9  }
```

The ConstantDepth behavior's primary parameter is the `depth` parameter on line 8. This behavior does make provisions for providing a range of depths, or a more gradually degrading utility function when this behavior is used to work with other behaviors reasoning about depth. In this mission however, the depth is mostly non-contentious between behaviors. The exception is when the PeriodicSurface behavior is active, in which case the priority weight for the PeriodicSurface behavior is simply set to suppress the ConstantDepth behavior.

10.3.3 Topic #3: The PeriodicSurface Behavior

The PeriodicSurface behavior, described generally in Section 8.5, is used in the Delta mission to bring the vehicle to the surface for GPS fixes periodically, to simulate the need for occasional navigation corrections. The behavior configuration, in file `delta.bhv` is shown in Listing 21 below. The first part of the configuration block, in lines 3-6, is for parameters defined generally for all IvP Helm behaviors, and the second part, in lines 8-13, is for parameters defined for the PeriodicSurface behavior.

Listing 21 - The PeriodicSurface behavior in the Delta mission, from the file delta.bhv.

```
0  //--------------------------------------------
1  Behavior = BHV_PeriodicSurface
2  {
3    name             = bhv_periodic_surface
4    pwt              = 1000
5    condition        = (MODE == LOITERING) or (MODE == RETURNING)
6    condition        = PSURFACE = true
7
8                period = 120
9       zero_speed_depth = 2
10   max_time_at_surface = 60
11          ascent_speed = 1.0
12          ascent_grade = fullspeed
13         mark_variable = GPS_UPDATE_RECEIVED
14 }
```

The PeriodicSurface behavior outputs an objective function solely on the *depth* decision variable. Note the priority weight in line 4 in Listing 21 is set to 1000, in comparison to the priority weight set for the ConstantDepth behavior on line 4 in Listing 20. When the PeriodicSurface behavior is running, it may or may not be active and producing an objective function over the *depth* decision variable. When it *is* active, the influence on *depth* simply overrules the influence from the ConstantDepth behavior due to these relative priority weights. For this reason, the ConstantDepth behavior may be conditioned on all active modes as in line 6 of Listing 20.

Note the parameter setting on line 13 above, setting the "mark variable" to `GPS_UPDATE_RECEIVED`. This is the default value, and the line could be safely be removed from the configuration block, but it is included anyway to be clear. The PeriodicSurface behavior, once it has noted that the vehicle has reached the surface, will wait until a value has been posted to `GPS_UPDATE_RECEIVED` that is different than its previous posting. Usually a timestamp suffices. The configuration of `uTimerScript` to achieve simulated GPS updates is discussed in more detail in Section 14.9.1.

10.3.4 Topic #4: The Waypoint Behavior with Survey Patterns

The Waypoint behavior, described generally in Section 8.1, is used in the Delta mission to conduct survey patterns like that shown in Figure 49. The behavior configuration, in file `delta.bhv` is shown in Listing 22 below. Note the survey pattern, described on line 15, does not list a set of points, but instead describes the pattern in terms of its properties such as the lane width and pattern orientation. This format is supported generally for building SegList objects from string specifications, discussed in Section 12.4.

Listing 22 - The Waypoint behavior in the Delta mission, from the file delta.bhv.

```
 0 //---------------------------------------------
 1 Behavior = BHV_Waypoint
 2 {
 3   name      = waypt_survey
 4   pwt       = 100
 5   condition = MODE==SURVEYING
 6   perpetual = true
 7   updates   = SURVEY_UPDATES
 8   endflag   = SURVEY = false
 9
10           lead = 8
11    lead_damper = 1
12          speed = 2.0   // meters per second
13         radius = 8.0
14         points = format=lawnmower, label=dudley_survey, x=80, y=-80,  \
                    width=70, height=30, lane_width=8, rows=north-south, \
                    degs=30
15 }
```

The waypoint behavior is configured to execute the survey pattern once, since the `repeat` parameter is not set and defaults to zero. Upon completion, it posts an end flag, configured on line 8, `SURVEY=false` to the MOOSDB. This immediately moves the vehicle out of the `SURVEYING` and into either the `LOITERING` or `RETURNING` mode depending on what it was doing when it was commanded to begin the survey. See the hierarchical mode declarations at the top of the `delta.bhv` file. Since the behavior is set with `perpetual=true` on line 6, completion of the survey pattern merely puts

the behavior in a virtual stand-by mode until it is given a new set of waypoints and it once again meets its run conditions.

The waypoint behavior is configured to accept dynamic parameter updates on line 7 through the MOOS variable SURVEY_UPDATES. In the Delta mission, updates are initiated by a mouse click in the pMarineViewer window, with the result being a post to the MOOSDB of the SURVEY_UPDATES variable. Such a posting would consist of a new value for the points parameter, replacing the initial configuration on line 14. This is discussed next.

10.3.5 Topic #4: Using pMarineViewer with Geo-referenced Mouse Clicks

In the Delta mission, the pMarineViewer application is used to post messages to the MOOSDB to (a) put the vehicle into the SURVEYING mode from either the LOITERING or RETURNING modes and (b) grab a user specified point in the operation area, from the user's mouse click, to be used as the center point of the commanded survey pattern. This situation is shown in Figure 49. This is achieved by utilizing the Mouse Context feature of pMarineViewer, discussed in Section **??**. The below two lines are inserted into the pMarineViewer configuration block in delta.moos:

```
left_context[survey-point] = SURVEY = true
left_context[survey-point] = SURVEY_UPDATES = points = vname=$(VNAME),  \
      x=$(XPOS), y=$(YPOS), format=lawnmower, label=delta, width=70,    \
      height=30, lane_width=8, rows=north-south, degs=80
```

Both lines declare that a MOOS variable-value pair is to be posted whenever a left mouse click is generated by the user. The fist line sets SURVEY=true, in an attempt to put the helm into the SURVEYING mode. This alone will not suffice to switch modes if the current value of the MOOS variable RETURN is set to true. See the mode declarations near the top of the delta.bhv file for this mission - a command to return overrides a command to perform a survey. The second line above associates with a left-mouse click a posting to the SURVEY_UPDATES variable. Recall from Listing 22 that this is the very variable through which the surveying waypoint behavior is configured to receive dynamic parameter updates. This posting is configured with a few macros, $(VNAME), $(XPOS), $(YPOS), which are filled in with the name and position of the current vehicle in the pMarineViewer window . Each user left-mouse click thus re-assigns the vehicle to a new survey pattern, and switches the helm mode, unless it has been commanded to return.

10.3.6 Suggestions for Further Experimenting with the Delta Mission

Failing to Reason about Depth

In this mission, *depth* is a mandatory helm decision variable, along with *course* and *speed*, as configured in delta.moos and shown in Listing 19. If a mission is not configured properly it's possible to bring about a helm mode where no behavior is reasoning about depth. Declaring *depth* to mandatory is equivalent to declaring that a situation where a helm iteration with no decision on *depth* is an error condition worth of an all-stop.

One way to bring this about in this mission, is to reconfigure the ConstantDepth behavior configuration in delta.bhv to replace line 6 in Listing 20 with the following

```
condition = (MODE==LOITERING)
```

By doing this, there will be no *depth* related behavior when the vehicle is in the RETURNING mode. Try re-running the mission. Note when the mission is first launched, the label next to the vehicle in pMarineViewer reads "dudley (PARK)(ManualOverride)". This is normal and indicates that the helm state is PARK, simply because the operator retains manual control. In addition to seeing this in the pMarineViewer window, this can also be confirmed by scoping on a few key helm variables including IVPHELM_STATE and IVPHELM_ALLSTOP:

```
$ uXMS delta.moos IVPHELM_STATE IVPHELM_ALLSTOP BHV_ERROR --show=source,time,community

  VarName               (S)ource    (T)ime      (C)ommunity   VarValue (MODE = SCOPE:PAUSED)
  ----------------      ----------  ----------  ----------     ----------- (61)
  IVPHELM_STATE          pHelmIvP    81.86       dudley        "PARK"
  IVPHELM_ALLSTOP        pHelmIvP    1.69        dudley        "ManualOverride"
  BHV_ERROR               n/a         n/a         n/a          n/a
  MODE                    n/a         n/a         n/a          n/a
```

Note that the IVPHELM_ALLSTOP value is posted once, until its value changes, and IVPHELM_STATE posts its value on every helm iteration regardless of the value. The latter variable also serves the purpose of a helm heartbeat indicator.

After noting the above, launch the mission by hitting the DEPLOY button, and note that the label next to the vehicle has changed to "dudley (LOITERING)". The four MOOS variables from above have also changed to:

```
  VarName               (S)ource    (T)ime      (C)ommunity   VarValue (MODE = SCOPE:PAUSED)
  ----------------      ----------  ----------  ----------     ----------- (103)
  IVPHELM_STATE          pHelmIvP    109.56      dudley        "DRIVE"
  IVPHELM_ALLSTOP        pHelmIvP    98.07       dudley        "clear"
  BHV_ERROR               n/a         n/a         n/a          n/a
  MODE                   pHelmIvP    98.07       dudley        "ACTIVE:LOITERING"
```

After the vehicle has been running a bit, try hitting the RETURN button which changes the helm to the RETURNING mode. In this case, due to our tinkering with the mission above, there is no behavior reasoning about depth, and the following message appears next to the vehicle on the screen: "dudley (Returning)(MissingDecVars:depth)". The four scoped MOOS variables also change to the following:

```
  VarName               (S)ource    (T)ime      (C)ommunity   VarValue (MODE = SCOPE:PAUSED)
  ----------------      ----------  ----------  ----------     ----------- (134)
  IVPHELM_STATE          pHelmIvP    621.34      dudley        "DRIVE"
  IVPHELM_ALLSTOP        pHelmIvP    617.09      dudley        "MissingDecVars:depth"
  BHV_ERROR              pHelmIvP    621.59      dudley        "MissingDecVars:depth"
  MODE                   pHelmIvP    617.09      dudley        "ACTIVE:RETURNING"
```

Note that, despite the error and all-stop event, the helm remains in drive, and may return to loitering or surveying at any time. Try configuring the helm in delta.moos to contain the line PARK_ON_ALLSTOP=true and note the difference in the above experiment. The generated error will bump the helm into the PARK helm status.

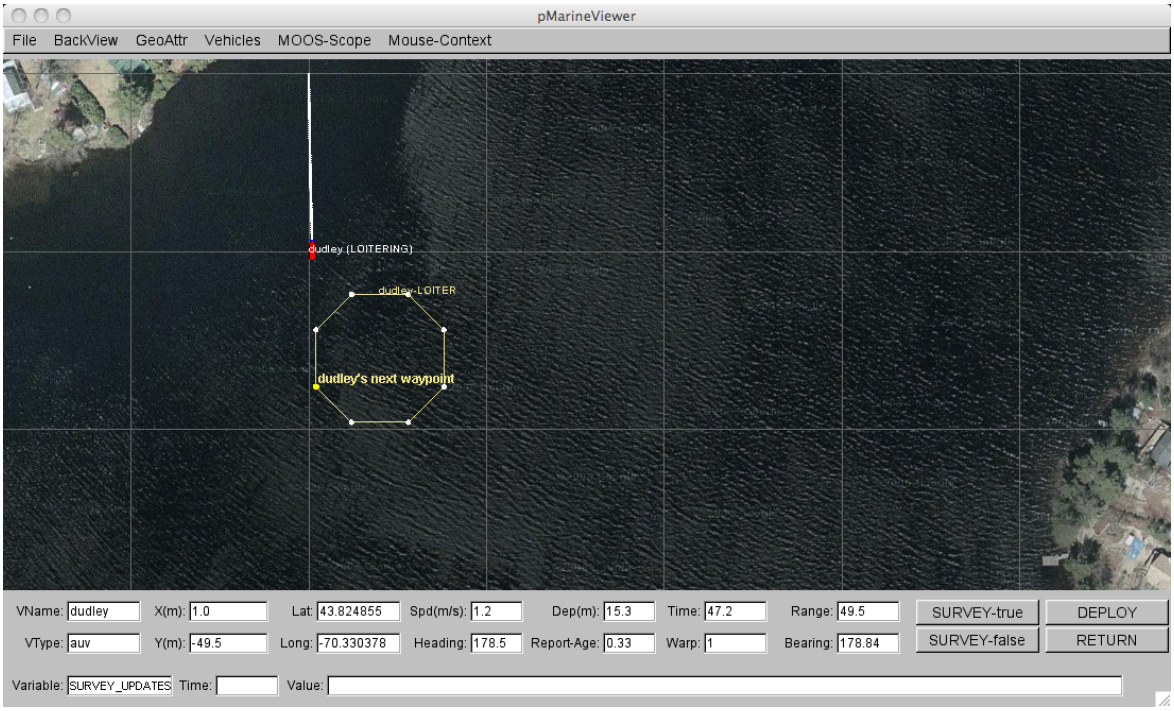


Figure 48: **The Delta Mission (1):** The vehicle `"dudley"` heads to its loiter region where it will remain indefinitely until it is commanded to return or perform a survey pattern. It will periodically surface for a GPS fix.

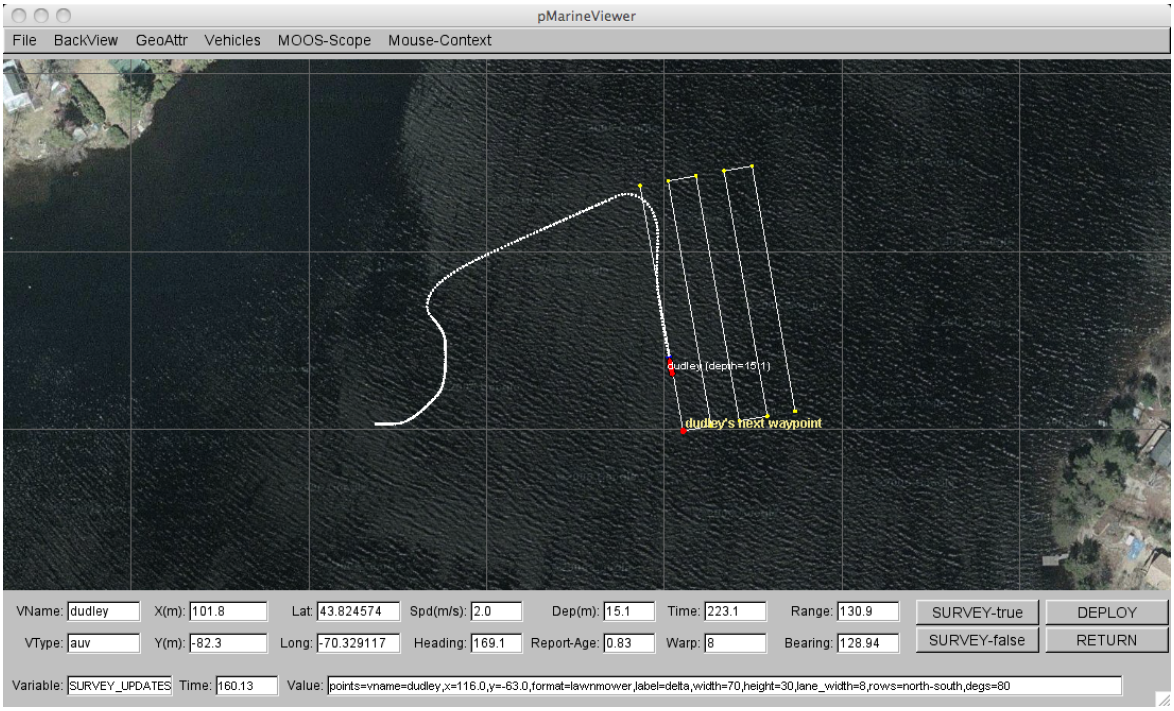


Figure 49: **The Delta Mission (2):** The user has clicked a point in the viewer around which a survey pattern is built. The vehicle exits the loitering mode and begins to execute the survey pattern.

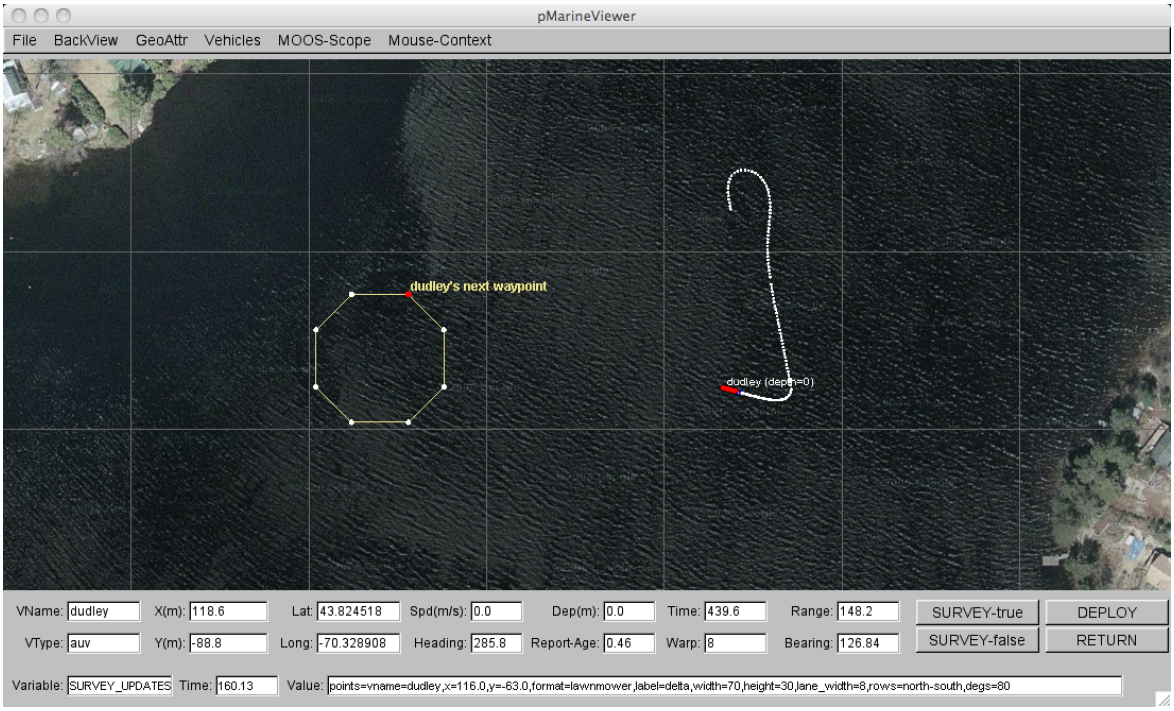


Figure 50: **The Delta Mission (3):** Once the vehicle has finished the survey pattern, it re-enters the loiter mode returning to its loiter region. It resumes periodic surfacing and immediately surfaces for a GPS fix.

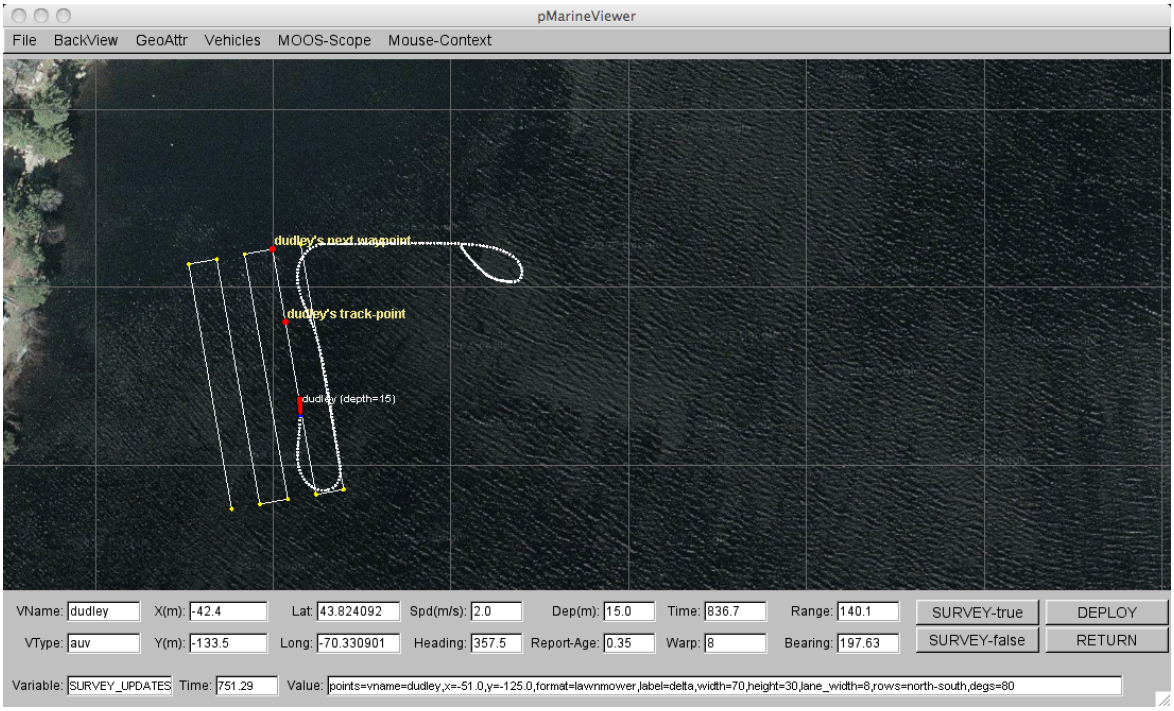


Figure 51: **The Delta Mission (4):** After resumption of loitering, the user clicks in the viewer a new point around which a new survey pattern is built. The entry point of the survey pattern automatically accommodates the vehicle.

10.4 Mission S5: The Echo Mission

The primary purpose of the Echo example mission is to illustrate the use of dynamically spawned behaviors. A simple behavior, the BearingLine behavior, is used to illustrate the idea. The BearingLine behavior simply posts a viewable point and viewable line segment representing the bearing from the present position of the vehicle to a fixed point in the operation area. Each new "bearing point" posted to the MOOSDB results in a newly spawned BearingLine behavior in the helm.

Overview of the Echo Mission Components and Topics

Mission:	"echo" in moos-ivp/missions/s5_echo
Behaviors:	BHV_Waypoint, BHV_BearingLine
MOOS Apps:	pHelmIvP, pLogger, uSimMarine, pMarinePID, pNodeReporter, pMarineViewer
Primary Topics:	(1) The BearingLine behavior. (2) Dynamic behavior spawning. (3) Sending updates to the original and spawned behaviors.
Side Topics:	(1) uTimerScript is used to auto-generate events with random components that lead to behavior spawning. (2) uHelmScope may be used to monitor the spawning and death of behaviors in the helm.

Launching the Echo Mission

The echo mission may be launched from the command line:

```
> cd moos-ivp/missions/s5_echo/
> ./launch.sh --warp=10
```

This should bring up a pMarineViewer window like that shown in Figure 52 on page 173, with a single vehicle, "henry", initially having the "PARK" helm state. After hitting the DEPLOY button in the lower right corner, the vehicle enters the "SURVEYING" mode and begins to proceed along the waypoints as shown.

10.4.1 Topic #1: The BearingLine Behavior

In Figure 52, note the line segment rendered from the vehicle in the direction of point (100, -100). This line segment is posted to the MOOSDB by the IvP Helm on behalf of the BearingLine behavior. This behavior does not influence the trajectory of the vehicle at all. The line segment acts as an easy visual confirmation that the behavior is instantiated and running properly. In the Echo mission this behavior is configured as in Listing 23. The bearing line originates from the present vehicle position toward the point specified on line 8. The length of the line is 50% of the present distance between the vehicle and the bearing point, as specified on line 7. The bearing point is also configured to be rendered by the configuration on line 9.

Listing 23 - Configuration of the BearingLine behavior in the Echo example mission.

```
0  //---------------------------------------------
1  Behavior = BHV_BearingLine
2  {
```

```
 3    name       = bng-line
 4    templating = clone
 5    updates    = BEARING_POINT
 6
 7         line_pct = 50
 8    bearing_point = 100,-100
 9          show_pt = true
10  }
```

The BearingLine behavior produces no (IvP) objective function, so by the definition of the behavior run states (Section 6.5.3), it is never in the *active* state. In the Echo mission, the BearingLine behavior is in the *running* state when the vehicle is surveying the five waypoints shown in Figure 52. This can be confirmed by launching a uHelmScope window:

```
uHelmScope moos-ivp/ivp/missions/s5_echo/echo.moos -x -p
```

The output should be similar to that shown in Listing 24 below. Note the Waypoint behavior is active, line 12, and the BearingLine behavior is in the running state, line 14.

Listing 24 - Output of the uHelmScope tool during the execution of the Echo mission.

```
 0  ==============   uHelmScope Report  ============== DRIVE  (8)
 1    Helm Iteration: 173     (hz=0.25)(5)  (hz=0.25)(100)  (hz=0.26)(max)
 2    IvP functions:  1
 3    Mode(s):
 4    SolveTime:      0.00    (max=0.01)
 5    CreateTime:     0.00    (max=0.01)
 6    LoopTime:       0.00    (max=0.01)
 7    Halted:         false   (0 warnings)
 8  Helm Decision: [speed,0,4,21] [course,0,359,360]
 9    course = 189.0
10    speed = 2.0
11  Behaviors Active: ---------- (1)
12    waypt_survey (43.0) (pwt=100.00) (pcs=6) (cpu=0.17)
13  Behaviors Running: --------- (1)
14    bng-line (43.0) (upd=0/0)
15  Behaviors Idle: ------------ (1)
16    waypt_return
17  Behaviors Completed: ------- (0)
18
19  #  MOOSDB-SCOPE ------------------------------------ (Hit '#' to en/disable)
20  @  BEHAVIOR-POSTS TO MOOSDB ---------------------- (Hit '@' to en/disable)
```

10.4.2 Topic #2 Dynamic Behavior Spawning

The Echo mission is configured to allow dynamic behavior spawning for the BearingLine behavior. Lines 4 and 5 in Listing 23 allow the spawning by configuring templating to be enabled on line 4, and specifying the MOOS variable through which spawning requests are received on line 5. Recall that templating can be enabled with either the `"clone"` or `"spawn"` options. In this case, `"clone"` option was chosen to allow the instantiation of one initial instance upon helm start-up, with the parameter configuration shown.

In this example, the pMarineViewer is configured to convert left-mouse clicks into posts of the BEARING_POINT variable to the MOOSDB, triggering the spawning of new BearingLine instances. A mouse click over the point (-5, -58) results in the post:

```
BEARING_POINT = "name=bng-line-5.0--158.0, bearing_point=-5.0,-158.0"
```

The helm receives mail for the BEARING_POINT variable since it registers automatically for each variable specified in any behavior configured with the UPDATES parameter. The helm examines this string before applying the update, and notes that the behavior name specified is unique (not currently instantiated) and rather than interpreting this as a request to update the existing BearingLine behavior already instantiated, interprets it as a request to spawn a new BearingLine instance if templating is enabled (which it is). The new behavior is spawned, with the behavior name specified. In this case the behavior name is based on the coordinates of the point clicked by the user. Two successive clicks on the same point will result in two posts to BEARING_POINT by pMarineViewer, but the second post will be effectively ignored by the helm. (It is read by the helm, but since the behavior name is one that is already known to the helm, the update is applied to that existing behavior instance. In this case such an update to the bearing_point parameter would be redundant.

After two such user mouse clicks, there will be two new BearingLine behaviors instantiated, and the situation would look similar to that shown in Figure 53. If the uHelmScope tool is still connected as above, the output would looks similar to that shown in Listing 25. Note the existence of three running BearingLine behavior instances reported in lines 14-16. The instance on line 14 was created upon helm startup, and the instances on lines 15-16 were created upon the user mouse clicks.

Listing 25 - Output of the uHelmScope tool during the later execution of the Echo mission.

```
 0  ==============   uHelmScope Report  ============== DRIVE  (12)
 1    Helm Iteration: 279     (hz=0.25)(5)  (hz=0.25)(100)  (hz=0.26)(max)
 2    IvP functions:  1
 3    Mode(s):        ACTIVE:SURVEYING
 4    SolveTime:      0.00    (max=0.01)
 5    CreateTime:     0.00    (max=0.01)
 6    LoopTime:       0.00    (max=0.01)
 7    Halted:         false   (0 warnings)
 8  Helm Decision: [speed,0,4,21] [course,0,359,360]
 9    course = 180.0
10    speed = 2.0
11  Behaviors Active: ---------- (1)
12    waypt_survey (69.6) (pwt=100.00) (pcs=4) (cpu=0.28)
13  Behaviors Running: --------- (3)
14    bng-line (69.6) (upd=0/0)
15    bng-line-5--58 (66.1) (upd=1/1)
16    bng-line13--124 (56.8) (upd=1/1)
17  Behaviors Idle: ------------ (1)
18    waypt_return
19  Behaviors Completed: ------- (0)
20
21  #  MOOSDB-SCOPE ------------------------------------ (Hit '#' to en/disable)
22  @  BEHAVIOR-POSTS TO MOOSDB ---------------------- (Hit '@' to en/disable)
```

10.4.3 Topic #3: Sending Updates to the Original and Spawned Behaviors

The action associated with the left-mouse click in the Echo mission is configured in the pMarineViewer configuration block in `echo.moos` by:

```
left_context[bng_point] = BEARING_POINT = name=bng-line$(X)-$(Y) # bearing_point=$(X),$(Y)
```

Setting the context for the left and right mouse clicks in pMarineViewer is discussed in Section **??** on page **??**. With the above configuration a left-mouse click may result in the following if clicked at the point `(-5, -58)`:

```
BEARING_POINT = name=bng-line-5--58 # bearing_point=-5,-58
```

Updating or changing the prevailing parameters for existing behaviors is possible via the use of the MOOS variable specified in the `UPDATES` parameter for each behavior. The only difference between an update that changes parameters of an existing behavior, and an update that spawns a new behavior is the inclusion of a `name=<behavior-name>` as above. If this component is present in the string posted to the MOOSDB and the `<behavior-name>` specifies an existing behavior, then that behavior only will have its parameters updated. If it does not specify an existing behavior, a new behavior will be spawned with the specified name. If the `name=<behavior-name>` component is not included in the posting, then the update will be applied to all behaviors configured to receive updates via that particular MOOS variable.

This case is a bit interesting since all newly spawned behaviors specify the same MOOS variable for receiving updates. Indeed a single poke to the MOOS variable `BEARING_POINT` could result in the simultaneous configuration modification of all instantiated BearingLine behaviors. In the Echo mission, pMarineViewer is configured to make the following pokes to the MOOSDB via the `ACTION` pull-down menu:

```
BEARING_POINT = show_pt=true
BEARING_POINT = show_pt=false
BEARING_POINT = line_pct=0
BEARING_POINT = line_pct=25
BEARING_POINT = line_pct=50
BEARING_POINT = line_pct=75
BEARING_POINT = line_pct=100
BEARING_POINT = "name=bng-line # line_pct=0"
BEARING_POINT = "name=bng-line # line_pct=50"
BEARING_POINT = "name=bng-line # line_pct=100"
```

The first two actions will turn on or off the rendering of the bearing points posted by each behavior. The next five actions will adjust the rendering length of the posted bearing line by each behavior. The last two actions will adjust the rendering length of the posted bearing line only for the behavior named `"bng-line"`, the behavior spawned at the time of helm start-up.

Suggestions for Tinkering

- After the mission is launched, use the ACTION pull-down menu in `pMarineViewer` to select `AUTO_SPAWN=true`. This enables a script via `uTimerScript` that automatically generates new bearing points, spawning new behaviors. These behaviors have their durations set randomly

by the script to be in range $[5, 10]$ seconds. Note the new bearing lines emerging and moving with the vehicle until the behavior dies. The script will lead to the spawning (and death) of 5000 behaviors over the course of about two hours of simulation at MOOSTimeWarp=1.

- With `AUTO_SPAWN=true` as above, let the vehicle return to the launch point (by clicking the `RETURN` button in the `pMarineViewer` window. After it reaches the return point and perhaps sits for a bit, hit the `DEPLOY` button once again to return the vehicle into its `SURVEYING` mode. Notice that initially there are many bearing lines rendered before returning to only a handful of bearing lines after a few seconds of simulation. This is because newly spawned behaviors do not start their duration clock until the first time the behavior enters the running state. All behaviors spawned while returning to the start point are effectively put on hold until the vehicle is re-deployed. Then they all start their duration clocks simultaneously and all die off 5-10 seconds later.

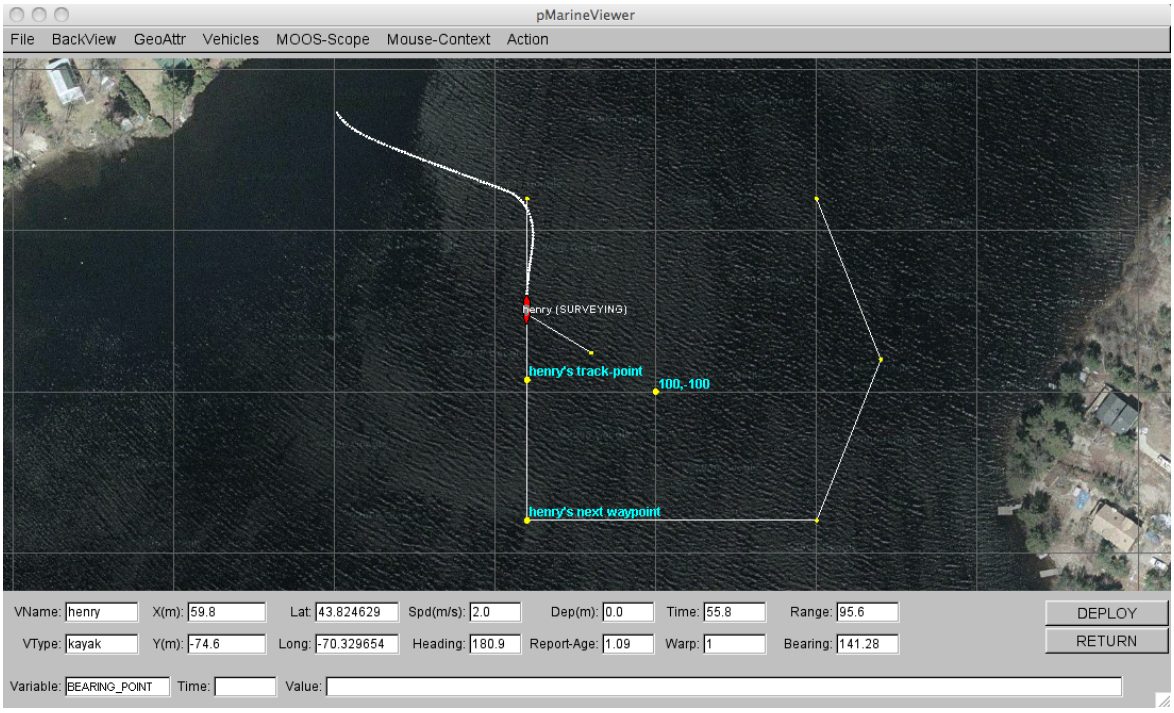


Figure 52: **The Echo Mission (1):** The vehicle `"henry"` traverses waypoints with the Waypoint behavior. The BearingLine behavior generates a viewable "bearing point" at `(100, -100)`, and a viewable "bearing line".

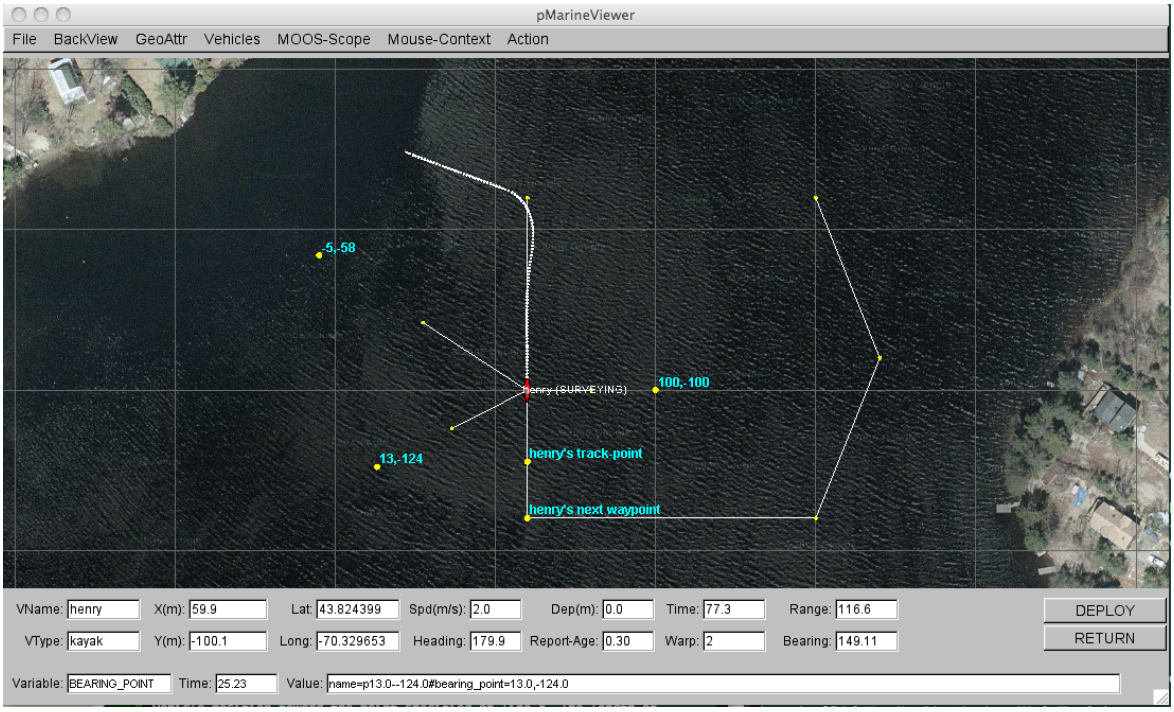


Figure 53: **The Echo Mission (2):** The user has clicked two new bearing points, `(-5, -58)`, and `(13, -124)`, and two new BearingLine behaviors have been spawned, each generating bearing line and bearing point visual outputs.

10.5 Mission S11: The Kilo Mission

The purpose of the Kilo mission is to illustrate the use of the standby helm, and the use of the TestFailure behavior. The standby helm is described in Section 5.7 and consists of an otherwise normally configured helm with the additional standby parameter invoked. This helm will wait in standby mode until the heartbeat of another primary helm ceases to be posted to the MOOSDB for some period of time. In this example mission, a primary and standby helm are configured with the primary helm executing a simple mission similar to the Alpha mission, but also using an instance of the TestFailure behavior. This behavior will be configured to trigger either a crash or hang of the primary helm to demonstrate the manner in which a standby helm will step in to take over with its own mission. The standby mission in this case is simply a behavior to return the vehicle to its launch position.

Overview of the Kilo Mission Components and Topics

Mission:	"kilo" in moos-ivp/missions/s11_kilo
Behaviors:	BHV_Waypoint, BHV_TestFailure
MOOS Apps:	pHelmIvP, pLogger, uSimMarine, pMarinePID, pNodeReporter, pMarineViewer
Primary Topics:	(1) The use of a standby helm. (2) The TestFailure behavior. (3) Scoping the the primary and standby helm states at runtime.
Side Topics:	(1) uHelmScope may be used to monitor the relationship between the shadow helm and primary helm.

Launching the Kilo Mission

The Kilo mission may be launched from the command line:

```
> cd moos-ivp/missions/s11_kilo/
> ./launch.sh --warp=10
```

This should bring up a pMarineViewer window like that shown in Figure 57 on page 179, with a single vehicle, "kilo", initially in the "PARK" mode. After hitting the DEPLOY button in the lower right corner, the vehicle enters the "SURVEYING" mode and begins to proceed along the waypoints as shown.

10.5.1 Topic #1: The Use of a Standby Helm

The vehicle in this example is configured with both a primary helm and a secondary helm. Configuring a mission with two helm instances is straight forward. The primary helm is configured as done normally, as in lines 3-11 in Listing 1 below. The standby helm is configure identically but with one additional line, as in line 23 in Listing 1 below. This line denotes the helm it to be a *standby* helm, and indicates the time threshold used in determining when to take over from a primary helm.

Listing 1: **kilo.moos:** Configuration of the primary and standby helm in the Kilo mission.

```
1  //-------------------- pHelmIvP config block
2
3  ProcessConfig = pHelmIvP
4  {
5    AppTick    = 4
6    CommsTick  = 4
7
8    Behaviors  = kilo.bhv
9    Domain     = course:0:359:360
10   Domain     = speed:0:4:21
11 }
12
13 //-------------------- pHelmIvP_Standby config block
14
15 ProcessConfig = pHelmIvP_Standby
16 {
17   AppTick    = 4
18   CommsTick  = 4
19
20   Behaviors  = kilo_standby.bhv
21   Domain     = course:0:359:360
22   Domain     = speed:0:4:21
23   STANDBY    = 2
24 }
```

Both helms are instances of the `pHelmIvP` application, but one of them is named `pHelmIvP_Standby`. The name of the application is not in any way used to put this helm instance into a standby role. It is chosen simply to be different from the primary helm, since the MOOSDB requires all connected applications to have unique names, and to be clear when or if debugging the mission.

The shadow helm is launched by `pAntler` with the name `pHelmIvP_Standby`. This is done by giving it the alternative name using the `pAntler ExtraProcessParams` parameter as shown on lines 15 and 18 below.

Listing 2: **kilo.moos:** Configuration of the primary and standby helm in the Kilo mission.

```
1  //--------------------------------------
2  // Antler configuration  block
3
4  ProcessConfig = ANTLER
5  {
6    MSBetweenLaunches = 200
7
8    Run = MOOSDB            @ NewConsole = false
9    Run = pLogger           @ NewConsole = false
10   Run = uSimMarine        @ NewConsole = false
11   Run = pNodeReporter     @ NewConsole = false
12   Run = pMarinePID        @ NewConsole = false
13   Run = pMarineViewer     @ NewConsole = false
14   Run = pHelmIvP          @ NewConsole = true
15   Run = pHelmIvP          @ NewConsole = true, ExtraProcessParams=HParams
16   Run = uProcessWatch     @ NewConsole = false
17
18   HParams=--alias=pHelmIvP_Standby
19 }
```

If the shadow helm is launched independently, not with `pAntler`, it may be launched with:

```
$ pHelmIvP kilo.moos --alias=pHelmIvP_Standby
```

10.5.2 Topic #2: The TestFailure Behavior

The Kilo mission uses the TestFailure behavior described in Section 8.13 to artificially generate a failure of the primary helm. In the Kilo mission it is configured to produce a "hung" helm with the following configuration:

Listing 3: **kilo.bhv:** Configuration of the BHV_TestFailure behavior.

```
1  //------------------------------------------
2  Behavior = BHV_TestFailure
3  {
4    name                = test_failure
5    condition           = DEPLOY=true
6    duration            = 120
7    duration_idle_decay = false
8
9          failure_type  = hang,3
10 }
```

The first four configuration parameters are defined for all IvP behaviors. A run condition of `DEPLOY=true` keeps this behavior idle until the mission is launched. Once it is launched, a countdown of 120 seconds begins until the behavior will hang. The `duration_idle_decay=false` setting ensures that the countdown doesn't begin until the behavior is in the running state. In this case the behavior will fail by hanging for three seconds. This will cause the primary helm to also appear to hang for three seconds, noted by a three second gap between heartbeats, defined by posting to the MOOS variable `IVPHELM_STATE`. Since the standby helm is configured to wait no longer than two seconds (see Listing 1), the standby helm will promptly take over control from the primary helm. The sequence of events in taking over control is described in more detail next.

10.5.3 Topic #3: Scoping the Helm State(s) at Runtime

The relationship between the primary and standby helm may be monitored by scoping on the variable `IVPHELM_STATE` during the course of the mission. Figure 54 below shows the situation shortly after the vehicle is deployed using the `uXMS` scoping tool.

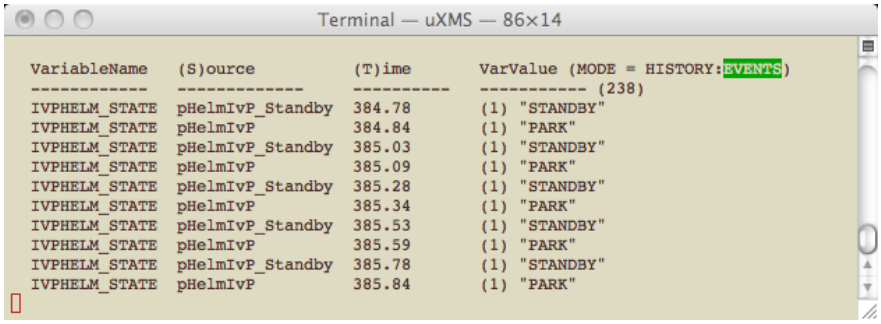


Figure 54: **Helm State in the Kilo Mission:** Both the standby and primary helms are operating normally, posting a helm state message about four times per second.

Initially the standby helm is showing a helm state of `"STANDBY"`, and the primary helm is showing a helm state of `"DRIVE"`. Since both helms are operating at the same frequency, they mostly alternate between postings as shown above.

The events shown in Figure 55 show that during the 9 identical postings ending at time 329.86, the standby helm is the only helm emitting a heartbeat. During this period the primary helm has either crashed or hung. Finally the secondary helm takes over and posts a helm state of `"DRIVE+"` for 107 consecutive heartbeats (iterations). Recall the `"+"` is to further distinquish that this helm was originally configured as a standby helm.

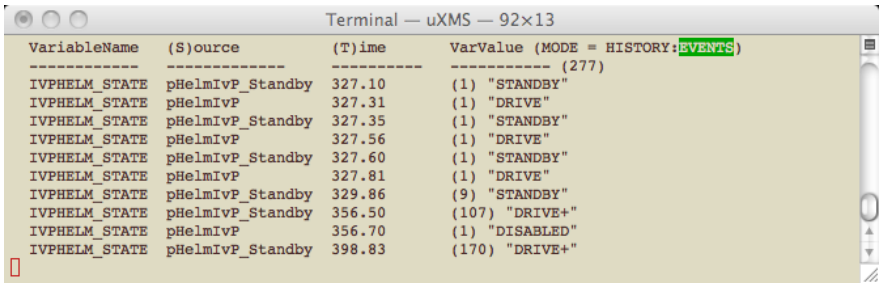


Figure 55: **Helm State in Kilo Mission after Standby Helm Takes Over Hung Primary Helm:** The standby helm has detected a delay in the primary helm heartbeat and takes over about two seconds later. Note the number in parentheses is the number of identical postings with the timestamp showing the time of the last of the identical postings.

Eventually it turns out that the original primary helm did not crash after all but was just temporarily hung. After emerging from its hung state, upon reading mail in the next loop, it realizes the standby helm has taken over and immediately concedes control and posts the `"DISABLED"` helm state, and will never again post. One may wonder why the primary helm, when it finally woke up, did not first mistakely post `"DRIVE"` before conceding control - after all it needs to first read its mail to learn that another helm is in control. The primary helm in fact did post `"DRIVE"` and `"DISABLED"` on two consecutive iterations. The first posting occurred (time 327.81) prior to querying the behaviors for input, and the second posting (time 356.70) occurred on the next iteration of upon reading mail.

In the case where the standby helm takes over for a crashed helm, the sequence of events, viewed from the perspective of posts to `IVPHELM_STATE`, is similar as shown in Figure 56. When

things are going fine the standby helm and primary helm alternately post helm states of `STANDBY` and `DRIVE` respectively as the first several posts below shoe. At some point the primary helm crashes and the only posts are made by the standby helm. In the example here, there are 10 `IVPHELM_STATE="STANDBY"` posts made while the standby helm is getting closer to triggering a take-over. Finally, upon takeover, it posts a helm state of `"DRIVE+"` thereafter. The primary helm is never heard from again.

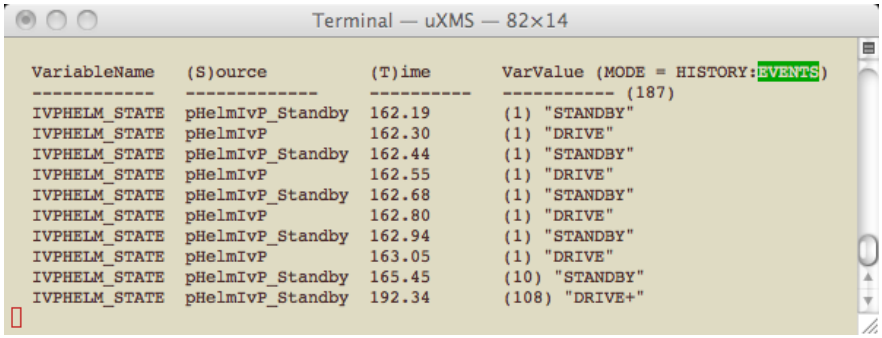


Figure 56: **Helm State in Kilo Mission after Standby Helm Takes Over Crashed Primary Helm:** The standby helm has detected a delay in the primary helm heartbeat and takes over about two seconds later. Note the number in parentheses is the number of identical postings with the timestamp showing the time of the last of the identical postings.

Suggestions for Tinkering

- *Analyze a helm takeover from the log files.* Run the Kilo mission again and, as opposed to monitoring things with a live scope as in Figures 54 - 56, let the mission play out. Change directories to the folder where the `.alog` file was created. Take a look at the `IVPHELM_STATE` postings using the `aloggrep` tool. Confirm that the sequence of events described in Figures 54 - 56 are consistent with entries of the log file. Hint: This can be achieved without mission configuration or code modifacations.

- *Crashing with helm with a mouse click.* Rather than waiting for the `TestFailure` behavior to run down its clock to failure, configure it to fail immediately upon entering the run state (`duration=0`). But also set an additional run-condition, e.g., `FAIL=true` and configure the `pMarineViewer` to make this posting upon a mouse-click. Test the modified Kilo mission by launching it, and clicking the mouse when ready to see the failure. This can be achieved solely through mission configuration modification, without code modification.

- *Configure a standby helm that finishes a task.* Create a primary and shadow helm where the primary helm is surveying a set of waypoints. The shadow helm should be able to survey the remaining points when/if the primary helm is taken over. Hint: this will likely require code generation or modification. Extra bonus if achievable solely through mission modification.

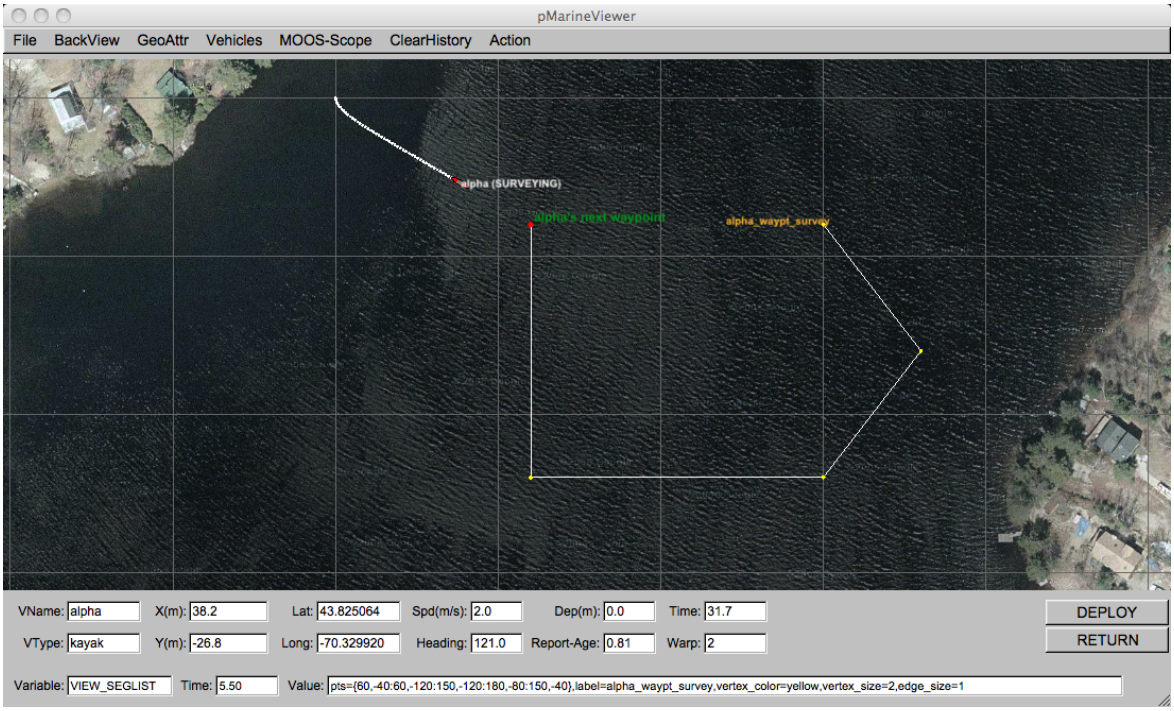


Figure 57: **The Kilo Mission (1):** The vehicle `"kilo"` traverses the waypoints using a primary helm, launched alongside a standby helm. The TestFailure behavior is a ticking bomb that will hang the primary helm momentarily.

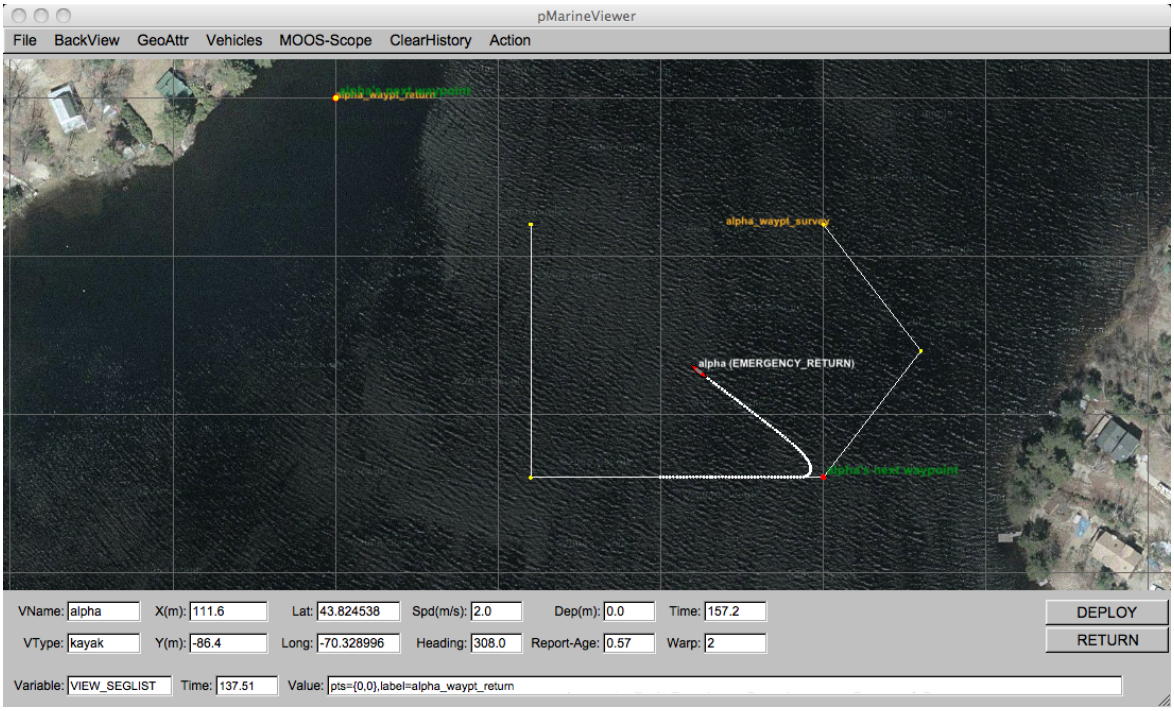


Figure 58: **The Kilo Mission (2):** The TestFailure behavior has hung the primary helm. The standby helm has detected the absence of a heartbeat (the `IVPHELM_STATE` variable), and take over with a simple return mission.

10.6 Mission M2: The Berta Mission

The Berta mission involves two vehicles repeatedly performing collision avoidance with one another. The primary capabilities highlighted are the helm's collision avoidance behavior, the basic contact manager (pBasicContactMgr), and the dynamic spawning and removal of behaviors related to contacts from the helm.

Overview of the Berta Mission Components and Topics

Mission:	"Berta" in moos-ivp/missions/m2_berta
Vehicles:	henry, gilda
Behaviors:	BHV_Loiter, BHV_AvoidCollision
MOOS Apps:	pHelmIvP, pLogger, uSimMarine, pBasicContactMgr, pMarinePID, pNodeReporter, pMarineViewer, uTimerScript, pMOOSBridge
Primary Topics:	(1) The IvP Helm AvoidCollision behavior / Dynamic behavior spawning. (2) Contact management with the pBasicContactMgr application.
Side Topics:	(1) uTimerScript is used in a command and control role for random permuting of vehicle loiter assignments. (2) pMOOSBridge is used for connecting one shoreside command and control community to two simulated vehicle communities. (3) The Loiter behavior is configured to receive dynamic updates on its loiter position.

Launching the Berta Mission

The *Berta* mission may be launched from the command line:

```
> cd moos-ivp/missions/m2_berta/
> ./launch.sh --warp=10
```

This should bring up a pMarineViewer window like that shown in Figure 59 on page 184, with two vehicles, "henry" and "gilda", each initially in the "PARK" state. After hitting the DEPLOY button in the lower right corner, the vehicle enters the "LOITERING" mode and begins to proceed along the loiter pattern as shown. The example mission is configured to periodically alter the position of each vehicle's loiter pattern, alternating between a random point in Region-A and Region-B shown in Figure 59. The user may click the PERMUTE-NOW button to immediately cause a new permutation and command sent to the vehicle, which otherwise happens automatically every three minutes.

10.6.1 Topic #1: The AvoidCollision Behavior and Dynamic Behavior Spawning

The AvoidCollision behavior runs on both vehicles and is configured identically on each vehicle in this mission. Its configuration is shown in Listing 26. When the helm first launches on each vehicle this behavior is not present. It is configured to be a dynamically spawned behavior - one for each known contact within a certain range of the vehicle. In this regard it depends on alerts from a contact manager running separately on each vehicle that posts the MOOS variable CONTACT_INFO, with the name of the new contact for which to spawn a new behavior.

The CONTACT_INFO variable is the variable for which the behavior is configured to receive dynamic updates, on line 6. The behavior is configured to allow dynamic instantiations in line 7 with

`templating = spawn`. The choice of `spawn` (vs. `clone`) means that a behavior of this configuration is not initially present unless cued via the `updates` interface. See the discussion on dynamic behavior spawning in Section 7.7, on page 93.

Listing 26 - Configuration of the AvoidCollision behavior in the Berta example mission.

```
 0  //------------------------------------------
 1  Behavior = BHV_AvoidCollision
 2  {
 3    name       = avd_collision
 4    pwt        = 200
 5    condition  = AVOID=true
 6    updates    = CONTACT_INFO
 7    endflag    = CONTACT_RESOLVED = $[CONTACT]
 8    templating = spawn
 9
10                contact = to-be-set
11       on_no_contact_ok = true
12            extrapolate = true
13                  decay = 30,60
14
15         pwt_outer_dist = 50
16         pwt_inner_dist = 20
17              pwt_grade = linear
18         completed_dist = 75
19      min_util_cpa_dist = 8
20      max_util_cpa_dist = 25
21    bearing_line_config = white:0, green:0.65, yellow:0.8, red:1.0
22  }
```

The parameters in lines 15-17 determine the level of activity of the behavior based on the range to the contact. The parameters in lines 15 and 16 refer to the priority weight of the behavior. Beyond 50 meters in range the weight will be 0% of its static weight of 200 assigned on line 4. (A weight of zero means that no objective function will be produced and the behavior is in the running state, but not the active state.) At 20 meters range, it will be at 100% of its static priority weight. The grade in priority between outer and inner distances is linear, due to the setting on line 17. The `min_util_cpa_dist`, and `max_util_cpa_dist` on lines 19 and 20 refer to the utilities assigned to candidate maneuvers in the objective function output by the behavior. Any candidate maneuver whose CPA is less than 8 meters will be given the lowest utility rating, equivalent to an actual collision. Likewise, any candidate maneuver whose CPA is greater than 25 will be given the highest utility rating, equivalent to missing the contact by infinite distance.

The `completed_dist`, on line 18, refers to the range to the contact beyond which the behavior will declare itself to be complete. Once completed, the behavior will post its end flags (none in this case), and will remove itself from the helm. In this regard, some coordination with the contact manager is needed. At the very least, the contact manager must be configured to post alerts to the mutually configured MOOS variable, `CONTACT_INFO`, on line 6 in Listing 26 and on line 16 in Listing 27. Note also that the `completed_dist` is set to 75 meters, which is higher than the 55 meters set for the `alert_range` in the contact manager. This means that, after the avoid collision behavior dies, the contact must come back within 55 meters before the contact manager generates another alert, triggering the helm to once again spawn a behavior for the contact.

Once a contact has gone out of range to the point where the AvoidCollision behavior dies, it typically never returns, in practice. There is no guarantee of this however, since the future maneuvers of either the contact or ownship may once again put them on a collision course. The Berta mission tests this scenario repeatedly. The two vehicles are configured to loiter in a loiter pattern that keeps them out range from each other in terms of triggering an AvoidCollision behavior in each of their helms (as in Figure 59). In this mission, a `uTimerScript` script is running in the shoreside community that periodically picks a new loiter location within one of the two regions shown in Figure 59. It picks a new location once every three minutes, alternating between the two regions for each vehicle each time. This ensures that there will be a reasonably unpredictable collision avoidance situation each time the vehicles transit to their new loiter locations. The permutations happen automatically once every three minutes, but the user may jump to the next permutation by clicking on the `PERMUTE-NOW` button.

In Figure 60, note the line segment rendered between the two vehicles. This line segment is posted to the MOOSDB by the IvP Helm on behalf of the AvoidCollision behavior. A white line is drawn between the two vehicles as soon as the AvoidCollision behavior is spawned. The line turns a non-white color as soon as the behavior begins to generate an objective function, thus actively influencing the helm to prefer maneuvers with a lower expected CPA. The line is initially green to indicate a lower behavior priority. As the priority grows, the line turns yellow and finally red when it is at the higher priorities. The behavior posting of these line segments is optional and configurable. The configuration on line 21 in Listing 26 is responsible for the postings in this mission.

In this example, the `extrapolate` option is turned on for the AvoidCollision behavior (line 12 in Listing 26). Extrapolation is typically enabled when long durations are experienced between updates on contact position (which is not the case here in simulation). The behavior will automatically use the linear extrapolated position given the contact's last known position, trajectory and time-stamp. The linear extrapolation will decay down to a zero speed (stationary solution) beginning at 30 seconds and coming to a complete stop at 60 seconds, due to the decay specification on line 13 of Listing 26.

10.6.2 Topic #2: Contact Management with pBasicContactMgr

In the Berta mission, the `pBasicContactMgr` is running on both simulated vehicles, configured identically with the configuration block shown in Listing 27. Both the contact manager and the helm are receiving `NODE_REPORT` messages containing information about contacts. The helm stores this information in its information buffer for any behavior that requests it during normal operation. The contact manager uses contact information to generate alerts, which potentially cues the helm to spawn new contact-related behaviors.

The contact manager may be configured to generate more than one alert. However, in the Berta mission, a single alert, on line 15 in Listing 27 is sufficient for spawning collision avoidance behaviors. Note the alert specifies a variable name, and variable value. The variable name, (`CONTACT_INFO`), is set to be the variable used for updates in the AvoidCollision behavior, on line 6 in Listing 26. The variable value is configured to be #-separated list of parameter-value pairs suitable as input to any behavior receiving updates. (See Section 7.2.2 on behavior updates.) In this case, the alert is configured to "update" the AvoidCollision behavior parameters `name`, and `contact`. Both

parameter values are comprised, in part, of the name of the contact, `$[VNAME]`. In the case of the behavior name, this is to ensure that the behavior name is unique, a condition for spawning a new behavior. It also ensures that multiple AvoidCollision behaviors will not be spawned for the same contact. In the case of the contact name, the behavior simply must know the name of the contact for which it avoiding a collision.

In the unlikely event that two distinct contacts are present in the field, and reporting the same vehicle name, it would be up to the contact manager to discern that these are indeed two contacts and not one. Currently this is beyond the algorithms implemented in `pBasicContactMgr`.

Listing 27 - Configuration of the pBasicContactMgr application in the Berta example mission.

```
0  //--------------------------------------------------------
1  // pBasicContactMgr MOOS Configuration Block
2
3  ProcessConfig = pBasicContactMgr
4  {
5    AppTick   = 2
6    CommsTick = 2
7
8    ALERT_RANGE     = 55
9    ALERT_CPA_RANGE = 70
10   ALERT_CPA_TIME  = 30
11   CONTACT_MAX_AGE = 3600
12   DISPLAY_RADII   = false
13   VERBOSE         = true
14
15   ALERT = var=CONTACT_INFO, val="name=avd_$[VNAME] # contact=$[VNAME]"
16 }
```

In the Berta example mission, the contact manager is configured to generate alerts when a know contact comes within 55 meters of the vehicle (line 8 of Listing 27). The contact manager also is configured to generate alerts when a contact comes within 70 meters (line 9) *and* its calculated CPA over next 30 seconds (line 10) falls within the alert range (again line 8). See Figure **??** on page **??** for more on the relationship between these two ranges.

The contact manager is capable of optionally posting to the MOOSDB two circles rendering the two ranges, the `ALERT_RANGE` and `ALERT_CPA_RANGE`. The rendering is turned off and on with the setting on line 12. The circles are posted under the variable `VIEW_POLYGON` in the local MOOSDBs on each vehicle, and each vehicle is configured to bridge this variable (via `pMOOSBridge`) to the shoreside community running the `pMarineViewer`.

Suggestions for Tinkering in the Berta Mission

- Alter the alert ranges in the contact manager, and enable the displaying of range radii (line 12 in Listing 27) and notice when the AvoidCollision behavior is spawned, by the presence of the bearing lines between the two contacts.
- Alter the `pwt_*_dist` parameters in the AvoidCollision behavior (lines 15-16 in Listing 26) and check how this affects the ranges between the vehicles as they avoid collisions.

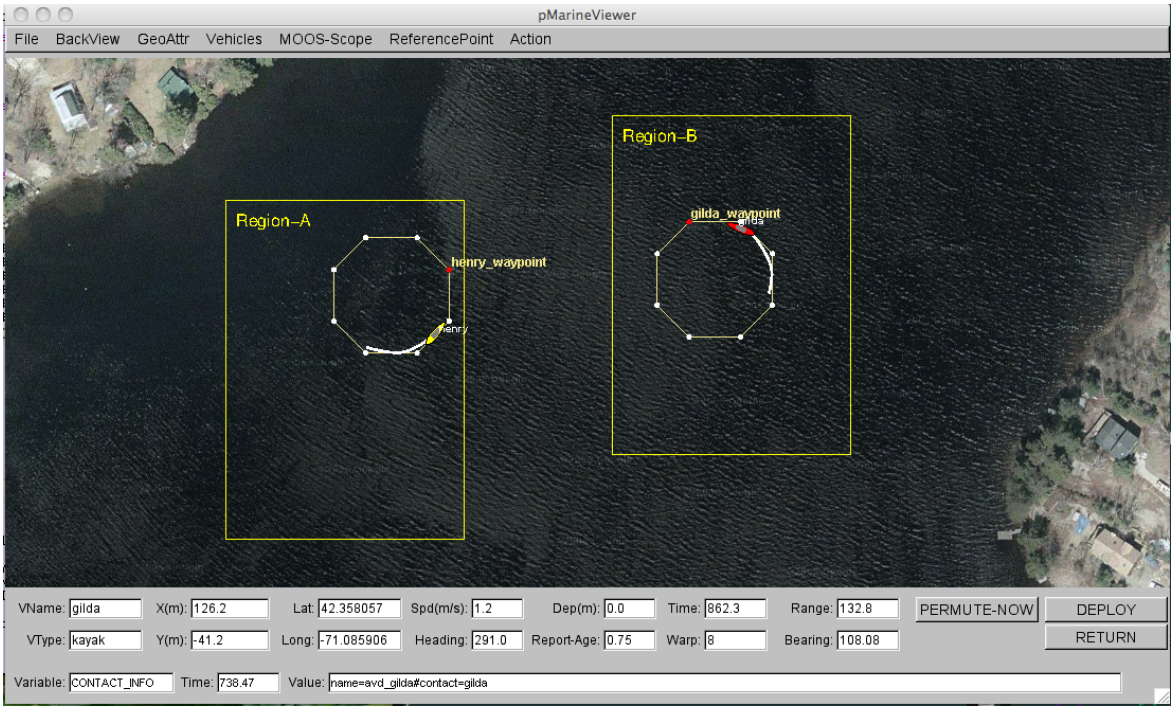


Figure 59: **The Berta Mission (1):** The two vehicles, `"henry"` and `"gilda"`, initially transit to their respective loiter stations and await further random location changes between the shown boxes.

Figure 60: **The Berta Mission (2):** The vehicles have changed their loiter stations and this puts them at risk for collision. The green bearing line between the vehicle indicates the presence of the AvoidCollision behavior.

11 uHelmScope: Scoping on the IvP Helm

11.1 Overview

The `uHelmScope` application is a console based tool for monitoring output of the IvP helm, i.e., the `pHelmIvP` process. The helm produces a few key MOOS variables on each iteration that pack in a substantial amount of information about what happened during a particular iteration. The helm scope subscribes for and parses this information, and writes it to a console window for the user to monitor. The user can dynamically pause or alter the output format to suit one's needs, and multiple scopes can be run simultaneously. The helm scope in no way influences the performance of the helm - it is strictly a passive observer.

The example console output shown in Listing 28 is used for explaining the `uHelmScope` fields.

Listing 28 - Example `uHelmScope` *output.*

```
 1  (alpha)(PAUSED)===========   uHelmScope Report  ============= DRIVE (133)
 2    Helm Iteration: 85
 3    IvP Functions:  1
 4    Mode(s):
 5    SolveTime:   0.00   (max=0.01)
 6    CreateTime:  0.00   (max=0.01)
 7    LoopTime:    0.00   (max=0.02)
 8    Halted:         false   (0 warnings)
 9  Helm Decision: [speed,0,4,21] [course,0,359,360]
10    speed = 2
11    course = 114
12  Behaviors Active: ---------- (1)
13    waypt_survey [21.29] (pwt=100) (pcs=6) (cpu=0.08) (upd=0/0)
14  Behaviors Running: --------- (0)
15  Behaviors Idle: ------------ (1)
16    waypt_return [always]
17  Behaviors Completed: ------- (0)
18
19
20  #  MOOSDB-SCOPE ------------------------------------ (Hit '#' to en/disable)
21  #
22  #  VarName            Source         Time  Commty  VarValue
23  #  -----------------  ------------  -----  ------  ----------------------
24  #  DEPLOY             pMari..iewer  25.05  alpha   "true"
25  #  IVPHELM_STATEVARS  pHelmIvP       5.42  alpha   "DEPLOY,MISSION,RETURN"
26  #  MISSION            n/a             n/a          n/a
27  #  RETURN             pMari..iewer  25.05  alpha   "false"                8
28
29
30  @  BEHAVIOR-POSTS TO MOOSDB ---------------------- (Hit '@' to en/disable)
31  @
32  @  MOOS Var      Behavior      Iter  Value
33  @  ------------  ------------  ----  ---------------------------------------
34  @  BHV_STATUS    waypt_return     1  name=waypt_return,p..te=idle,updates=n/a
35  @  ------------  ------------  ----  ---------------------------------------
36  @  CYCLE_INDEX   waypt_survey     1  0
37  @  VIEW_POINT    waypt_survey     1  x=60,y=-40,active=f..r=red,vertex_size=4
38  @  VIEW_SEGLIST  waypt_survey     1  pts={60,-40:60,-160.._size=4,edge_size=1
39  @  WPT_INDEX     waypt_survey     1  0
40  @  WPT_STAT      waypt_survey    84  vname=alpha,behavio..es=0,dist=30,eta=15
```

There are three groups of information in the `uHelmScope` output on each report to the console - the general helm overview (lines 1-17), a `MOOSDB` scope for a select subset of MOOS variables (lines

20-27), and a report on the MOOS variables published by the helm on the current iteration (lines 30-40). The output of each group is explained in the next three subsections.

11.2 The Helm Summary Section of the uHelmScope Output

The first block of output produced by `uHelmScope` provides an overview of the helm. This is lines 1-17 in Listing 28, but the number of lines may vary with the mission and state of mission execution. This block is virtually identical to the appcast report generated by the helm itself. So another way of doing `uHelmScope` style scoping is with an appcast viewing tool (`uMAC`, `uMACView`, and `pMarineViewer`). But with these tools, you would only see part of the information found in `uHelmScope`. The MOOSDB-Scope and Behaviors-Post portion of of `uHelmScope` is not part of the appcast report posted by the helm.

11.2.1 The Helm Status (Lines 1-8)

The integer value at the end of line 1 indicates the number of `uHelmScope` reports written to the console. This can confirm to the user that an action that should result in a new report generation has indeed worked properly. The integer on line 2 is the counter kept by the helm, incremented on each helm iteration. The value on Line 3 represents the the number of IvP functions produced by the active helm behaviors, one per active behavior. The solve-time on line 5 represents the time, in seconds, needed to solve the IvP problem comprised the n IvP functions. The number that follows in parentheses is the maximum solve-time observed by the scope. The create-time on line 6 is the total time needed by all active behaviors to produce their IvP function output. The loop time on line 7 is simply the sum of lines 5 and 6.

The Boolean on line 8 is true only if the helm is halted on an emergency or critical error condition. Also on line 8 is the number of warnings generated by the helm. This number is reported by the helm and *not* simply the number of warnings observed by the scope. This number coincides with the number of times the helm writes a new message to the variable `BHV_WARNING`.

11.2.2 The Helm Decision (Lines 9-11)

The helm decision space, i.e., IvP domain, is displayed on line 9. Each decision variable is given by its name, low value, high value, and the number of decision points. So `[speed,0,4,21]` represents values {`0, 0.25, 0.5, ..., 3.75, 4.0`}. The following lines used to display the actual helm decision. Occasionally the helm may be configured with one of its decision variables configured to be *optional*. The helm may not produce a decision on that variable on some iteration if no behaviors are reasoning about that variable. In this case the label `"varbalk"` may be shown next to the decision variable to indicate no decision.

11.2.3 The Helm Behavior Summary (Lines 12-17)

Following this is a list of all the active, running, idle and completed behaviors. At any point in time, each instantiated IvP behavior is in one of these four states and each behavior specified in the behavior file should appear in one of these groups. Technically all *active* behaviors are also *running* behaviors but not vice versa. So only the running behaviors that are not active (i.e., the behaviors that could have, but chose not to produce an objective function), are listed in the

"Behaviors Running:" group. Immediately following each behavior the time, in seconds, that the behavior has been in the current state is shown in parentheses. For the active behaviors (see line 13) this information is followed by the priority weight of the behavior, the number of pieces in the produced IvP function, and the amount of CPU time required to build the function. If the behavior also is accepting dynamic parameter updates the last piece of information on line 13 shows how many successful updates where made against how many attempts. A failed update attempt also generates a helm warning, counted on line 8. The idle and completed behaviors are listed by default one per line. This can be changed to list them on one long line by hitting the 'b' key interactively.

11.3 The MOOSDB-Scope Section of the uHelmScope Output

A built-in generic scope function is built into uHelmScope, not different in style from uXMS. The scope ability in uHelmScope provides two advantages: first, it is simply a convenience for the user to monitor a few key variables in the same screen space. Second, uHelmScope automatically registers for the variables that the helm reasons over to determine the behavior activity states. It will register for all variables appearing in behavior conditions, runflags, activeflags, inactiveflags, endflags and idleflags. It will also register for variables involved in the helm hierarchical mode definitions. The list of these variables is provided by the helm itself when it publishes IVPHELM_STATEVARS.

For example, the output in Listing 28 was derived from scoping on the alpha mission, and launching from the terminal with:

```
$ uHelmScope alpha.moos IVPHELM_STATEVARS
```

In this case the variable IVPHELM_STATEVARS itself is added to the scope list, and the *value* of this variable contains the three other variables on the scope list, reported by the helm to be involved in conditions or flags. The MISSION variable has not been written to because MISSION="complete" is the endflag of the return behavior in the alpha mission. At the point where this snapshot was taken, this behavior had not completed.

The lines comprising the MOOSDB-Scope section of the uHelmScope output are all preceded by the '#' character. This is to help discern this block from the others, and as a reminder that the whole block can be toggled off and on by typing the '#' character. The columns in Listing 28 are truncated to a set maximum width for readability. The default is to have truncation turned on. The mode can be toggled by the console user with the 't' character, or set in the MOOS configuration block or with a command line switch. A truncated entry in the VarValue column has a '+' at the end of the line. Truncated entries in other columns will have ".." embedded in the entry. Line 24 shows an example of both kinds of truncation.

The variables included in the scope list can be specified in the uHelmScope configuration block of a MOOS file. In the MOOS file, the lines have the form:

```
var = <MOOSVar>, <MOOSVar>, ...
```

An example configuration is given in Listing 32. Variables can also be given on the command line. Duplicates requests, should they occur, are simply ignored. Occasionally a console user may want to suppress the scoping of variables listed in the MOOS file and instead only scope on a couple variables given on the command line. The command line switch -c will suppress the variables listed

in the MOOS file - unless a variable is also given on the command line. In line 26 of Listing 28, the variable `MISSION` is a *virgin* variable, i.e., it has yet to be written to by any MOOS process and shows `n/a` in the five output columns. By default, virgin variables are displayed, but their display can be toggled by the console user by typing `'v'`.

11.4 The Behavior-Posts Section of the uHelmScope Output

The Behavior-Posts section is the third group of output in `uHelmScope`. It lists MOOS variables and values posted by the helm. Each variable was posted by a particular helm behavior and the grouping in the output is accordingly by behavior. Unlike the variables in the MOOSDB-Scope section, entries in this section only appear if they were written by the helm. The lines comprising the Behavior-Posts section of the `uHelmScope` output are all preceded by the '@' character. This is to help discern this block from the others, and as a reminder that the whole block can be toggled off and on by typing the '@' character. As with the output in the MOOSDB-Scope output section, the output may be truncated. A value that has been truncated will contain the `".."` characters around the middle of the string as in lines 34, 37-38, and 40.

11.5 Console Key Mapping and Command Line Usage

User input is accepted at the console during a `uHelmScope` session, to adjust either the content or format of the reports. It operates in a couple different *refresh* modes. In the *paused* refresh mode, after a report is posted to the console no further output is generated until the user requests it. In the *streaming* refresh mode, new helm summaries are displayed as soon as they are received. The refresh mode is displayed in the report on the very first line as in Listing 28.

The key mappings can be summarized in the console output by typing the 'h' key, which also sets the refresh mode to *paused*. The key mappings shown to the user are shown in Listing 29.

Listing 29 - Key mapping summary shown after hitting 'h' in a console.

```
 1  KeyStroke  Function
 2  ---------  ---------------------------------------------
 3  Getting Help:
 4      h      Show this Help msg - 'r' to resume
 5
 6  Modifying the Refresh Mode:
 7     Spc     Refresh Mode: Pause (after updating once)
 8      r      Refresh Mode: Streaming (throttled)
 9      R      Refresh Mode: Streaming (unthrottled)
10
11  Modifying the Content Mode:
12      d      Content Mode: Show normal reporting (default)
13      w      Content Mode: Show behavior warnings
14      l      Content Mode: Show life events
15      m      Content Mode: Show hierarchical mode structure
16
17  Modifying the Content Format or Filtering:
18      b      Toggle Show Idle/Completed Behavior Details
19      `      Toggle truncation of column output
20      v      Toggle display of virgins in MOOSDB-Scope output
21      #      Toggle Show the MOOSDB-Scope Report
22      @      Toggle Show the Behavior-Posts Report
```

```
23
24  Hit 'r' to resume outputs, or SPACEBAR for a single update
```

Several of the same preferences for adjusting the content and format of the `uHelmScope` output can be expressed on the command line, with a command line switch. Command line usage is shown in Listing 30, and may be obtained from the command line by invoking:

```
$ uHelmScope --help
```

Listing 30 - Command line usage of uHelmScope.

```
 1  ================================================================
 2  Usage: uHelmScope file.moos [OPTIONS] [MOOS Variables]
 3  ================================================================
 4
 5  Options:
 6    --alias=<ProcessName>
 7        Launch uHelmScope with the given process name rather
 8        than uHelmScope.
 9    --clean, -c
10        MOOS variables specified in given .moos file are excluded
11        from the MOOSDB-Scope output block.
12    --example, -e
13        Display example MOOS configuration block.
14    --help, -h
15        Display this help message.
16    --interface, -i
17        Display MOOS publications and subscriptions.
18    --noscope,-x
19        Suppress MOOSDB-Scope output block.
20    --noposts,-p
21        Suppress Behavior-Posts output block.
22    --novirgins,-g
23        Suppress virgin variables in MOOSDB-Scope output block.
24    --streaming,-s
25        Streaming (unpaused) output enabled.
26    --trunc,-t
27        Column truncation of scope output is enabled.
28    --version,-v
29        Display the release version of uHelmScope.
30
31  MOOS Variables
32    MOOS_VAR1 MOOSVAR_2, ..., MOOSVAR_N
33
34  Further Notes:
35    (1) The order of command line arguments is irrelvant.
36    (2) Any MOOS variable used in a behavior run condition or used
37        in hierarchical mode declarations will be automatically
38        subscribed for in the MOOSDB scope.
```

The command line invocation also accepts any number of MOOS variables to be included in the MOOSDB-Scope portion of the `uHelmScope` output. Any argument on the command line that does not end in `.moos`, and is not one of the switches listed above, is interpreted to be a requested MOOS variable for inclusion in the scope list. Thus the order of the switches and MOOS variables

do not matter. These variables are added to the list of variables that may have been specified in the uHelmScope configuration block of the MOOS file. Scoping on *only* the variables given on the command line can be accomplished using the -c switch. To support the simultaneous running of more than one uHelmScope connected to the same MOOSDB, uHelmScope generates a random number N between 0 and 10,000 and registers with the MOOSDB as uHelmScope_N.

11.6 Helm-Produced Variables Used by uHelmScope

There are six variables published by the helm to which uHelmScope subscribes. These provide critical information for generating uHelmScope reports.

The first two variables, IVPHELM_STATE and IVPHELM_SUMMARY are published on each iteration of the helm. The former is published regardless of the helm state. This variable serves as the helm heartbeat. The latter is only published when the helm is in the DRIVE state. The below examples give a feel for the content:

```
IVPHELM_STATE    = "DRIVE"
IVPHELM_SUMMARY  = "iter=66,ofnum=1,warnings=0,utc_time=1209755370.74,solve_time=0.00,
                   create_time=0.02,loop_time=0.02,var=speed:3.0,var=course:108.0,
                   halted=false,running_bhvs=none,
                   active_bhvs=waypt_survey$6.8$100.00$1236$0.01$0/0,
                   modes=MODE@ACTIVE:SURVEYING,idle_bhvs=waypt_return$55.3$n/a,
                   completed_bhvs=none"
```

The IVPHELM_SUMMARY variable contains all the dynamic information included in the general helm overview (top) section of the uHelmScope output. It is a comma-separated list of var=val pairs. The helm publishes this in a journal style, ommitting certain content if they are unchanged between iterations. When uHelmScope launches, it publishes to the variable IVPHELM_REJOURNAL which the helm interprets as a request to send a full-content message on the next iteration, before resuming journaling.

The IVPHELM_LIFE_EVENT is posted only when a behavior is spawned or dies. For missions without dynamic behavior spawning, this variable will only be posted upon startup for each initial static behavior. Note that the helm only publishes life events as they occur, so if the helm scope is launched after the helm, earlier events may not be reflected in the life event report. The below example gives a feel for the content of this variable:

```
IVHELM_LIFE_EVENT = "time=814.09, iter=3217, bname=bng-line-bng-line132--104,
                    btype=BHV_BearingLine, event=spawn,
                    seed=name=bng-line132--104#bearing_point=132,-104"
```

The IVPHELM_DOMAIN, IVPHELM_STATEVARS, and IVPHELM_MODESET variables are typically only produced once, upon startup.

```
IVPHELM_DOMAIN    = "speed,0,4,21:course,0,359,360"
IVPHELM_STATEVARS = "RETURN,DEPLOY"
IVPHELM_MODESET   = "---,ACTIVE#---,INACTIVE#ACTIVE,SURVEYING#ACTIVE,RETURNING"
```

The IVP_DOMAIN variable also contributes to this section of output by providing the IvP domain used by the helm. The IVPHELM_STATEVARS variable affects the MOOSDB-Scope section of the uHelmScope output by identifying which MOOS variables are used by behaviors in conditions, runflags, endflags and idleflags.

11.7 Configuration Parameters for uHelmScope

Configuration for uHelmScope amounts to specifying a set of parameters affecting the terminal output format. An example configuration is shown in Listing 32, with all values set to the defaults. Launching uHelmScope with a MOOS file that does not contain a uHelmScope configuration block is perfectly reasonable.

Listing 11.31: Configuration Parameters for uHelmScope.

behaviors_concise:	If true, the idle and completed behaviors are reported all on one line rather than separate lines. Legal values: true, false. The default is true.
display_bhv_posts:	If true, the behavior-posts section of the report is shown. This can also be toggled at run time with the '@' key. Legal values: true, false. The default is true. Section 11.4.
display_moos_scope:	If true, the MOOS variable scope section of the report is shown. This can also be toggled at run time with the '#' key. Legal values: true, false. The default is true. Section 11.3.
display_virgins:	If true, variables in the MOOS scope section of the report is shown will be shown even if they have never been written to. This can also be toggled at run time with the 'g' key. Legal values: true, false. The default is true. Section 11.3.
paused:	If true, uHelmScope launches in the paused mode. Legal values: true, false. Default value is true.
tuncated_output:	If true, output in the MOOS-scope or behavior-scope section of the report is truncated. This can also be toggled at run time with the '‘' key. Legal values: true, false. The default is false.
var:	A comma-separated list of variables to scope on in the MOOS-Scope block. Multiple lines may be provided. Section 11.3.

An example configuration file may be obtained from the command line with:

```
$ uHelmScope --example or -e
```

This will show the output shown in Listing 32 below.

Listing 32 - Example configuration of the uHelmScope application.

```
1  ============================================================
2  uHelmScope Example MOOS Configuration
3  ============================================================
```

```
 4
 5  ProcessConfig = uHelmScope
 6  {
 7    AppTick   = 1    // MOOSApp default is 4
 8    CommsTick = 1    // MOOSApp default is 4
 9
10    paused    = false   // default
11
12    display_moos_scope = true     // default
13    display_bhv_posts  = true     // default
14    display_virgins    = true     // default
15    truncated_output   = false    // default
16    behaviors_concise  = true     // default
17
18    var  = NAV_X, NAV_Y, NAV_SPEED, NAV_DEPTH   // MOOS vars are
19    var  = DESIRED_HEADING, DESIRED_SPEED       // case sensitive
20  }
```

Each of the parameters can also be set on the command line, or interactively at the console, with one of the switches or keyboard mappings listed in Section 11.5. A parameter setting in the MOOS configuration block will take precedence over a command line switch.

11.8 Publications and Subscriptions for uHelmScope

The interface for `uHelmScope`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uHelmScope --interface
```

11.8.1 Variables Published by uHelmScope

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section **??**.
- IVPHELM_REJOURNAL: A request to the helm to rejournal its summary output. Section 11.6.

11.8.2 Variables Subscribed for by uHelmScope

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section **??**.
- <USER-DEFINED>: Variables identified for scoping by the user in the `uHelmScope` will be subscribed for. See Section 11.3.
- <HELM-DEFINED>: As described in Section 11.3, the variables scoped by `uHelmScope` include any variables involved in the preconditions, runflags, idleflags, activeflags, inactiveflags, and endflags for any of the behaviors involved in the current helm configuration.
- IVPHELM_LIFE_EVENT: A description of a helm life event, the birth or death of a behavior and the manner in which it was spawned. See Section 11.6.
- IVPHELM_SUMMARY: A comprehensive summary of the helm status including behavior status summaries and most recent helm decision. See Section 11.6.

- IVPHELM_STATEVARS: A helm-produced list of MOOS variables involved in the logic of determining behavior activation. Any variable involved in mode conditions or behavior conditions. See Section 11.6.
- IVPHELM_DOMAIN: The specification of the IvP Domain presently used by the helm. See Section 11.6.
- IVPHELM_MODESET: A description of the helm's hierarchical mode specification. See Section 11.6.
- IVPHELM_STATE: A short description of the helm state: either STANDBY, PARK, DRIVE, or DISABLED. See Section 11.6.

12 Geometry Utilities

12.1 Brief Overview

This section discusses a few geometry data structures often used by the helm and the pMarineViewer application - convex polygons, lists of line segments, points and vectors. These data structures are implemented by the classes `XYPolygon`, `XYSegList`, `XYPoint`, and `XYVector` respectively in the `lib_geometry` module distributed with the MOOS-IvP software bundle. The implementation of these class definitions is somewhat shielded from the helm user's perspective, but they are often involved in parameter settings of for behaviors. So the issue of how to specify a given geometric structure with a formatted string is discussed here.

Furthermore, the pMarineViewer application accepts these data structures for rendering by subscribing to three MOOS variables `VIEW_POLYGON`, `VIEW_SEGLIST`, `VIEW_POINT`, anv `VIEW_VECTOR`. These variables contain a string format representation of the structure, often with further visual hints on the color or size of the edges and vertices for rendering. These variables may originate from any MOOS application, but are also often posted by helm behaviors to provide visual clues about what is going on in the vehicle. In the Alpha mission described in Section 4 for example, the waypoint behavior posted a seglist representing the set of waypoints for which it was configured, as well as posting a point indicating the next point on the behavior's list to traverse.

12.2 General Geometric Object Properties

Each of the four geometric objects, `XYPolygon`, `XYSegList`, `XYPoint`, and `XYVector`, are a subclass of the general `XYObject` class, and share certain properties discussed here.

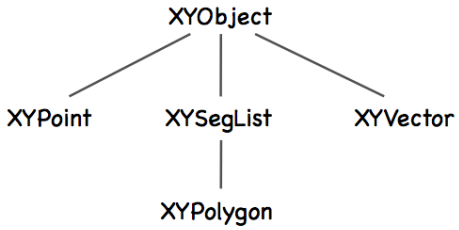


Figure 61: **Class hierarchy:** All geometric objects are subclasses of the `XYObject` class.

Common Properties

The following properties are defined at the `XYObject` level. All properties may be optionally left undefined by the user.

- *label*: A string that is often rendered in a GUI alongside the object rendering. In the `pMarineViewer` application, the label is also used as a unique identifier and successive receipt of geometric objects with the same label will result in the new one "replacing" the older one. Thus the visual effect of "moving" objects is rendered in this way.

- *msg*: A string typically regarded as an alternative to the label string when rendering an object. In the `pMarineViewer` application, if an object has a non-null *msg* field, this will be used for rendering the label instead of the *label* field.
- *type*: A string conveying some information on the type of the object from the source application. For example a point may be of type "waypoint" or "rendez-vous", and so on.
- *time*: The time field is a double that may optionally be set to indicate when the point was generated, or how long it should exist before expiring, or however an application may wish to interpret it.
- *source*: A string representing the source that generated the object. This may distinguish MOOS applications or helm behaviors for example.

Rendering Hint Properties

The following optional properties are defined at the `XYObject` level for providing hints to applications that may be using them for rendering.

- *active*: This is a Boolean that is regarded as `true` if left unspecified, and is used to indicate whether or not the object should be rendered. By setting this value to `false` for a given object, it would effectively be erased in the `pMarineViewer` application for example.
- *vertex_size*: This non-negative floating point value is a hint for how large to draw a vertex point for the given object. If left unspecified in the `pMarineViewer` application for example, the size of the vertex would be determined by the global setting for vertices set in the GUI.
- *edge_size*: This non-negative floating point value is a hint for how wide to draw an edge for the given object (if it has edges). If left unspecified in the `pMarineViewer` application for example, the width of the edge would be determined by the global setting for edges set in the GUI.
- *vertex_color*: This parameter specifies a hint or request to draw the vertices in this object in the given color specification. Colors may be specified by any defined color string or RGB string value as described in Appendix C. In the `pMarineViewer` application, if this hint is not provided for received objects, the vertex color would be determined by the global setting for vertices set in the GUI. If the color is set to `"invisible"`, this is effectively a request that the vertex not be rendered.
- *edge_color*: This parameter specifies a hint or request to draw the edgees in this object in the given color specification. Colors may be specified by any defined color string or RGB string value as described in Appendix C. In the `pMarineViewer` application, if this hint is not provided for received objects, the edge color would be determined by the global setting for edges set in the GUI. If the color is set to `"invisible"`, this is effectively a request that the edge not be rendered.
- *label_color*: This parameter specifies a hint or request to draw the label for this object in the given color specification. Colors may be specified by any defined color string or RGB

string value as described in Appendix C. In the `pMarineViewer` application, if this hint is not provided for received objects, the label color would be determined by the global setting for labels set in the GUI. If the color is set to `"invisible"`, this is effectively a request that label not be rendered.

12.3 Points

Points are implemented in the `XYPoint` class, and minimally represent a point in the x-y plane. These objects are used internally for applications and behaviors, and may also be involved in rendering in a GUI and therefore may have additional fields to support this as described in Section 12.2.

12.3.1 String Representations for Points

The only required information for a point specification is its position in the x-y plane. A third value may optionally be specified in the z-plane. If z is left unspecified, it will be set to zero. The standard string representation is a comma-separated list of param=value pairs like the following examples:

```
point   = x=60, y=-40
point   = x=60, y=-40, z=12
point   = z=19, y=5, x=23
```

Partly for backward compatibility, a very simplified string representation of a point is also supported:

```
point   = 60,-40
point   = 60, -40, 0
```

An example of string representation of point with all the optional parameters described in Section 12.2 might look something like:

```
point   = x=60, y=-40, label=home, label_color=red, source=henry, type=waypoint,
          time=30, active=false, vertex_color=white, vertex_size=5, msg=bingo
```

Note that although a few different string formats are supported for *specifying* a point, only a single format is used when a `XYPoint` object is serialized into a string representation.

12.4 Seglists

Seglists are implemented in the `XYSegList` class and are comprised of an ordered set of vertices, implying line segments between each vertex. Seglist instances may be used for many things, but perhaps most often used to represent a set of vehicle waypoints. The geometry library has a number of basic operations defined for this class such as intersection testing with other objects, rotation, proximity testing and so on.

12.4.1 Standard String Representation for Seglists

The only requirement for a seglist specification is one or more vertex locations in the x, y plane. Values may optionally be specified in the z plane. If z is left unspecified it will default to zero. The standard string representation is a comma-separated list of param=value pairs like the following examples:

```
points    = pts={60,-40:60,-160:150,-160:180,-100:150,-40},label=foxtrot,type=top
```

Note the double-colon separator between the last pair of points as above. By "standard" format we mean it is not only accepted as initialization input, but is also the format produced when an existing instance is serialized using the object's `string XYSeglist::get_spec()` function. If z values are used, an example may look like the following which is the same as the above but associates $z = 2$ with each point:

```
points = pts={60,-40,2:60,-160,2:150,-160,2:180,-100,2:150,-40,2},label=foxtrot
```

An example of a string representation with all the optional parameters described in Section 12.2 might look something like:

```
points = {pts=60,-40,2:60,-160,2:150,-160,2:180,-100,2:150,-40,2},label=foxtrot
         label_color=green, source=henry, type=return_path, time=30, active=false,
         vertex_color=white, vertex_size=5, edge_size=2, edge_color=red, msg=bingo
```

12.4.2 The Lawnmower String Representation for Seglists

Seglists may also be built using the `lawnmower` format. The following is an example:

```
points = format=lawnmower, label=foxtrot, x=0, y=40, height=60, width=180,
         lane_width=15, rows=north-south, degs=45
```

The rotation of the pattern can optionally be specified in radians. For example, `degs=45` is equivalent to `rads=0.785398`. If, for some reason, both are specified, the seglist will be built using the `rads` parameter.

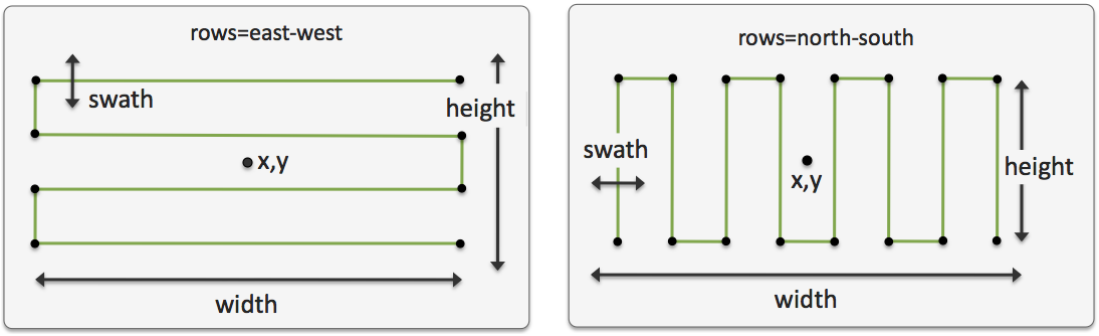


Figure 62: **Seglists built with the `lawnmower` format:** The pattern is specified by (a) the location of the center of the pattern, (b) the height and width of the pattern, (c) the lane width which determines the number of rows, (d) whether the pattern rows proceed north-south or east-west, and (e) an optional rotation of the pattern.

12.4.3 Seglists in the pMarineViewer Application

The `pMarineViewer` application registers for the MOOS variable `VIEW_SEGLIST`. The viewer maintains a list of seglists keyed on the label field of each incoming seglist. A seglist received with a thus far unique label will be added to the list of seglists rendered in the viewer. A seglist received with a non-unique label will replace the seglist with the same label in the memory of the viewer. This has the effect of erasing the old seglist since each iteration of the viewer redraws everything, the background and all objects, from scratch several times per second.

The label is the only text rendered with the seglist in `pMarineViewer`. Since the label is also used as the key, if the user trys to "update" the label, or use the label to convey changing information, this may inadvertently result in accumulating multiple seglists in the viewer, each drawn over one another. Instead, the `VIEW_SEGLIST` may be posted with the message to be posted in the `msg=value` component. When this component is non-null, `pMarineViewer` renders it instead of the contents in the label component.

12.5 Polygons

Polygons are implemented in the `XYPolygon` class. This implementation accepts as a valid construction only specifications that build a convex polygon. Common operations used internally by behaviors and other applications, such as intersection tests, distance calculations etc, are greatly simplified and more efficient when dealing with convex polygons.

12.5.1 Standard String Representation for Polygons

The only requirement for a polygon specification is three or more vertex locations in the x, y plane constituting a convex polygon. Values may optionally be specified in the z plane. If z is left unspecified it will default to zero. The standard string representation is a comma-separated list of param=value pairs like the following examples:

```
points   = pts={60,-40:60,-160:150,-160:150,-40},label=foxtrot,type=one
```

Note the double-colon separator between the last pair of points as above. By "standard" format we mean it is not only accepted as initialization input, but is also the format produced when an existing instance is serialized using the object's `string XYPolygon::get_spec()` function. If z values are used, an example may look like the following which is the same as the above but associates $z = 4$ with each point:

```
points   = pts={60,-40,4:60,-160,4:150,-160,4:150,-40,4},label=foxtrot,type=one
```

Polygons are defined by a set of vertices and the simplest way to specify the points is with a line comprised of a sequence of colon-separated pairs of comma-separated x-y points in local coordinates such as:

```
polygon   = 60,-40:60,-160:150,-160:180,-100:150,-40:label,foxtrot
```

If one of the pairs, such as the last one above, contains the keyword `label` on the left, then the value on the right, e.g., `foxtrot` as above, is the label associated with the polygon. An alternative notation for the same polygon is given by the following:

```
polygon    = label=foxtrot, pts="60,-40:60,-160:150,-160:180,-100:150,-40"
```

This is an comma-separated list of equals-separated pairs. The ordering of the comma-separated components is insignificant. The points describing the polygon are provided in quotes to signify to the parser that everything in quotes is the right-hand side of the `pts=` component. Both formats are acceptable specifications of a polygon in a behavior for which there is a `polygon` parameter.

12.5.2 A Polygon String Representation using the Radial Format

Polygons may also be specified by their shape and the shape parameters. For example, a commonly used polygon is formed by points of an equal radial distance around a center point. The following is an example:

```
polygon    = format=radial, label=foxtrot, x=0, y=40, radius=60, pts=6, snap=1
```

The `snap` component in the above example signifies that the vertices should be rounded to the nearest 1-meter value. The `x`, `y` parameters specify the middle of the polygon, and `radius` parameters specify the distance from the center for each vertex. The `pts` parameters specifies the number of vertices used, as shown in Figure 63.

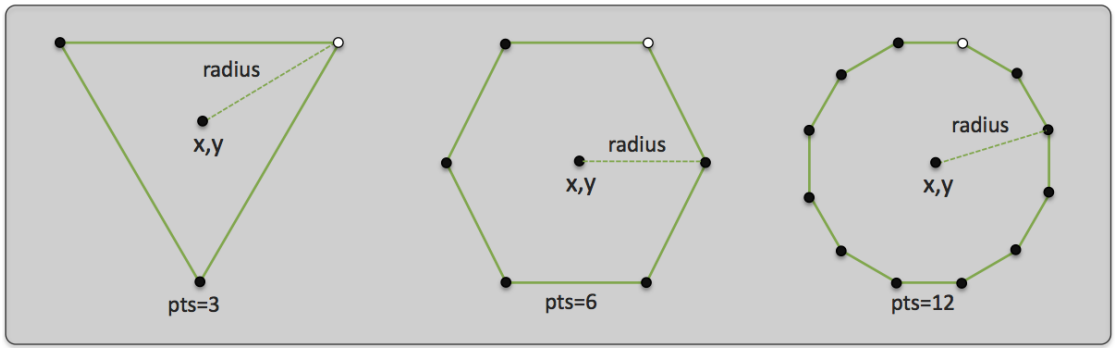


Figure 63: **Polygons built with the `radial` format:** Radial polygons are specified by (a) the location of their center, (b) the number of vertices, and (c) the radial distance from the center to each vertex. The lighter vertex in each polygon indicates the first vertex if traversing in sequence, proceeding clockwise.

12.5.3 A Polygon String Representation using the Ellipse Format

Polygons may also be built using the `ellipse` format. The following is an example:

```
polygon    = label=golf, format=ellipse, x=0, y=40, degs=45, pts=14, snap=1,
             major=100, minor=70
```

The `x`, `y` parameters specify the middle of the polygon, the `major` and `minor` parameters specify the radial distance of the major and minor axes. The `pts` parameters specifies the number of vertices used, as shown in Figure 64.

The rotation of the ellipse can optionally be specified in radians. For example, `degs=45` is equivalent to `rads=0.785398`. If, for some reason, both are specified, the polygon will be built using the `rads` parameter. When using the ellipse format, a minimum `pts=4` must be specified.

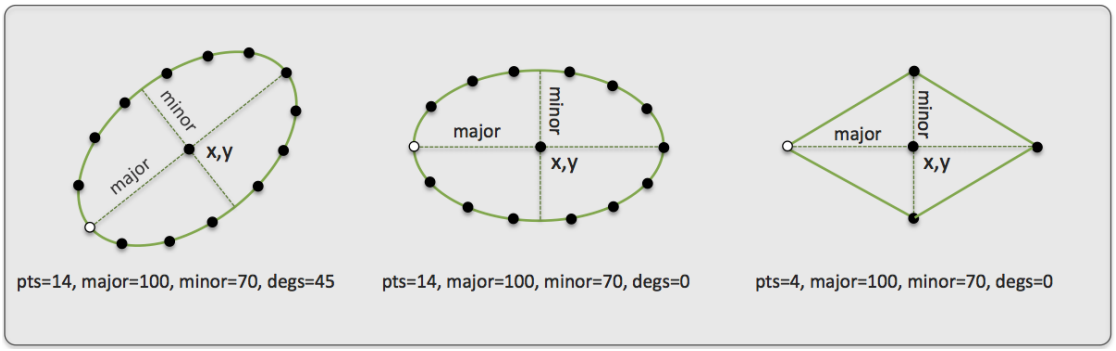


Figure 64: **Polygons built with the `ellipse` format:** Ellipse polygons are specified by (a) the location of their center, (b) the number of vertices, (c) the length of their major axis, (d) the length of their minor axis, and (e) the rotation of the ellipse.The lighter vertex in each polygon indicates the first vertex if traversing in sequence, proceeding clockwise.

12.5.4 Optional Polygon Parameters

Polygons also may have several optional fields associated with them. The `label` field is string that is often rendered with a polygon in MOOS GUI applications such as the pMarineViewer. The `label_color` field represents a color preference for the label rendering. The `type` and `source` fields are additional string fields for further distinguishing a polygon in applications that handle them. The `active` field is a Boolean that is used in the pMarineViewer application to indicate whether the the polygon should be rendered. The `time` field is a double that may optionally be set to indicate when the polygon was generated, or how long it should exist before "expiring", or however an application may wish to interpret it. The `vertex_color`, `edge_color`, and `vertex_size` fields represent further rendering preferences. The following are two equivalent further string representations:

```
polygon = format=radial, x=60, y=-40, radius=60, pts=8, snap=1, label=home,
          label_color=red, source=henry, type=survey, time=30, active=true,
          vertex_color=white, vertex_size=5, edge_size=2
polygon = format,radial:60,-40:radius,60:pts,8:snap,1:label,home:
          label_color,red:source,henry:type,survey:time,30:active,true:
          vertex_color,white:vertex_size,5:edge_size,2
```

The former is a more user-friendly format for specifying a polygon, perhaps found in a configuration file for example. The latter is the string representation passed around internally when `XYPolygon` objects are automatically converted to strings and back again in the code. This format is more likely to be found in log files or seen when scoping on variables with one of the MOOS scoping tools.

12.6 Vectors

Vectors are implemented in the `XYVector` class and are essentially comprised of a location, direction and magnitude. They may be used in certain applications dealing with simulated or sensed forces, or simply used in a GUI application to render current fields or an instantaneous vehicle pose and

trajectory. Minimally each vector consists of a location in the x, y plane and a direction and magnitude. They may also be configured with all relevant drawing hints discussed in Section 12.2.

12.6.1 String Representations for Vectors

Vector objects may be initialized from a string representation, and converted to a string representation from an existing object. The following is an example:

```
vector = x=5,y=10,ang=45,mag=20
```

Alternatively, instead of expressing the vector in terms of its direction and magnitude, it may also be given in terms of its magnitude in both the x and y direction. The following vector is nearly identical, modulo rounding errors, to the above configured vector:

```
vector = x=5,y=10,xdot=12.142,ydot=12.142
```

When a vector object is serialized to a string by invoking the object's native serialization function, the first of the two above formats will be used. The following is a string example using many of the general object fields and drawing hints available:

```
vector = x=5,y=10,ang=45,mag=20,label=pingu,source=simulator,type=wind,
         vertex_size=2,vertex_color=red,edge_color=red,edge_size=2,head_size=12
```

The last parameter, `"head_size=12"`, is a drawing hint unique to the vector object and refers to how big the arrow head should be rendered. If left unspecified, it may simply be rendered at whatever size the GUI application uses by default.

12.6.2 Vectors in the pMarineViewer Application

The `pMarineViewer` application registers for the MOOS variable `VIEW_VECTOR`. The viewer maintains a list of vectors keyed on the label field of each incoming vector. A vector received with a thus far unique label will be added to the list of vectors rendered in the viewer. A vector received with a non-unique label will replace the vector with the same label in the memory of the viewer. This has the effect of erasing the old vector since each iteration of the viewer redraws everything, the background and all objects, from scratch several times per second.

The label is the only text rendered with the vector in `pMarineViewer`. Since the label is also used as the key, if the user trys to "update" the label, or use the label to convey changing information, this may inadvertently result in accumulating multiple vectors in the viewer, each drawn over one another. Instead, the `VIEW_VECTOR` may be posted with the message to be posted in the `msg=value` component. When this component is non-null, `pMarineViewer` renders it instead of the contents in the label component.

Exercise 12.1: Poking a vector for visualising in pMarineViewer.

- Try running the Alpha mission again from Section 4. The window should look similar to Figure 6. The `uPokeDB` tool is described in [4].
- Poke the MOOSDB with:

  ```
  $ uPokeDB alpha.moos VIEW_VECTOR=x=100,y=0,mag=40,ang=135,label=one
  ```

  A vector should appear in the viewer 100 meters East of the vechicle's starting position.
- Poke the MOOSDB with:

  ```
  $ uPokeDB alpha.moos VIEW_VECTOR=x=100,y=-40,mag=40,ang=135,label=two
  vertex_color=orange,vertex_size=12,edge_size=5,edge_color=green,head_size=12
  ```

  A new vector should appear in the viewer 40 meters South of the first vector, with all the drawing hints from the command line.
- Poke the MOOSDB with:

  ```
  $ uPokeDB alpha.moos VIEW_VECTOR=x=120,y=0,mag=40,ang=135,label=one
  ```

  The first vector poked will appear to have moved 20 meters to the East. Since it has the same label as the first vector, the second vector effectively replaces the first one in pMarineViewer's memory.

13 pMarineViewer: A GUI for Mission Monitoring and Control

13.1 Overview

The `pMarineViewer` application is a MOOS application written with FLTK and OpenGL for rendering vehicles and associated information and history during operation or simulation. A screen shot of a simple one-vehicle mission is shown below in Figure 65.

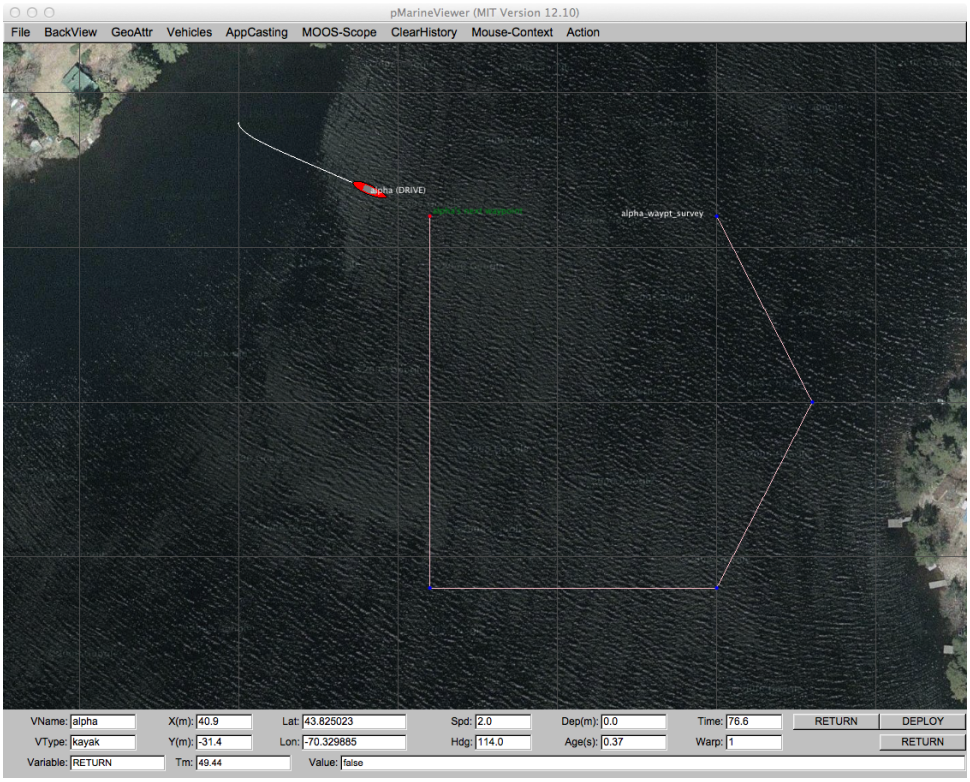


Figure 65: A `pMarineViewer` screen-shot executing a simple one-vehicle mission. The track of the vehicle is shown along with the set of waypoints it will traverse during this mission.

The user is able manipulate a geo display to see multiple vehicle tracks and monitor key information about individual vehicles. In the primary interface mode the user is a passive observer, only able to manipulate what it sees and not able to initiate communications to the vehicles. However there are hooks available and described later in this section to allow the interface to accept field control commands. With Release 12.11, appcasting viewing is supported to allow the `pMarineViewer` user to view appcasts across multiple fielded vehicles within a single optional window pane. This is described more fully in Section 13.5.

13.1.1 The Shoreside-Vehicle Topology

In some simple simulation single-vehicle arrangements `pMarineViewer` may co-exist in the same MOOS community as the helm and other components of a simulated vehicle. This is the case in the Alpha example mission. A more typical module topology, however, is that shown in Figure

66, where pMarineViewer is running in its own dedicated local MOOS community while simulated vehicles, or real vehicles on the water, transmit information in the form of a stream of *node reports* to the local community.

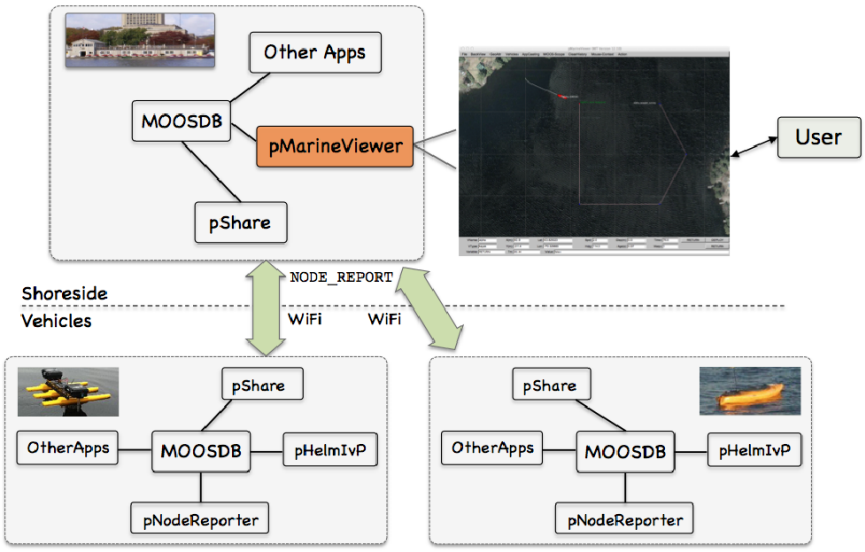


Figure 66: A common usage of the pMarineViewer is to have it running in a local MOOSDB community while receiving node reports on vehicle poise from other MOOS communities running on either real or simulated vehicles. The vehicles can also send messages with certain geometric information such as polygons and points that the view will accept and render.

A key variable subscribed to by pMarineViewer is the variable NODE_REPORT, which has the following structure given by an example:

```
NODE_REPORT = "NAME=henry,TYPE=uuv,TIME=1195844687.236,X=37.49,Y=-47.36,SPD=2.40,
              HDG=11.17,LAT=43.82507169,LON=-70.33005531,TYPE=KAYAK,MODE=DRIVE,
              ALLSTOP=clear,index=36,DEP=0,LENGTH=4"
```

Reports from different vehicles are sorted by their vehicle name and stored in histories locally in the pMarineViewer application. The NODE_REPORT is generated by the vehicles based on either sensor information, e.g., GPS or compass, or based on a local vehicle simulator.

In addition to node reports, pMarineViewer subscribes to several other types of information typically originating in the individual vehicle communities. This include several types of geometric shapes for which pMarineViewer has been written to handle. This includes points, polygons, lists of line segments, grids and so on. This is described further in Section 13.3.

In addition to consuming the above information, pMarineViewer may also be configured to post certain information, usually for command and control purposes. Since this is mission-specific, this information is completely configured by the user to suit the mission. Posted information may also be tied to mouse clicks to allow, for example, a vehicle to be deployed to a point clicked by the users. This is described further in Section 13.1.5.

13.1.2 Description of the pMarineViewer GUI Interface

The viewable area of the GUI has three parts as shown in Figure 67 below. In the upper right, there is a geo display area where vehicles and perhaps other objects are rendered. The blue panes on the upper left displays appcast information. These panes hold appcast output from any appcast-enabled MOOS application running on any node, including the shoreside node. This is a new feature of Release 12.11 and may be toggled off and on with the `'a'` key, and may be configured to be either open or closed by setting the `appcast_viewable` parameter inside the `pMarineViewer` MOOS configuration block.

In the lower pane, certain data fields associated with the *active* vehicle are updated. Multiple vehicles may be rendered simultaneously, but only one vehicle, the *active* will be reflected in the data fields in the lower pane. Changing the designation of which vehicle is active can be accomplished by repeatedly hitting the 'v' key. The active vehicle is always rendered as red, while the non-active vehicles have a default color of yellow. Individual vehicle colors can be given different default values (even red, which could be confusing) by the user.

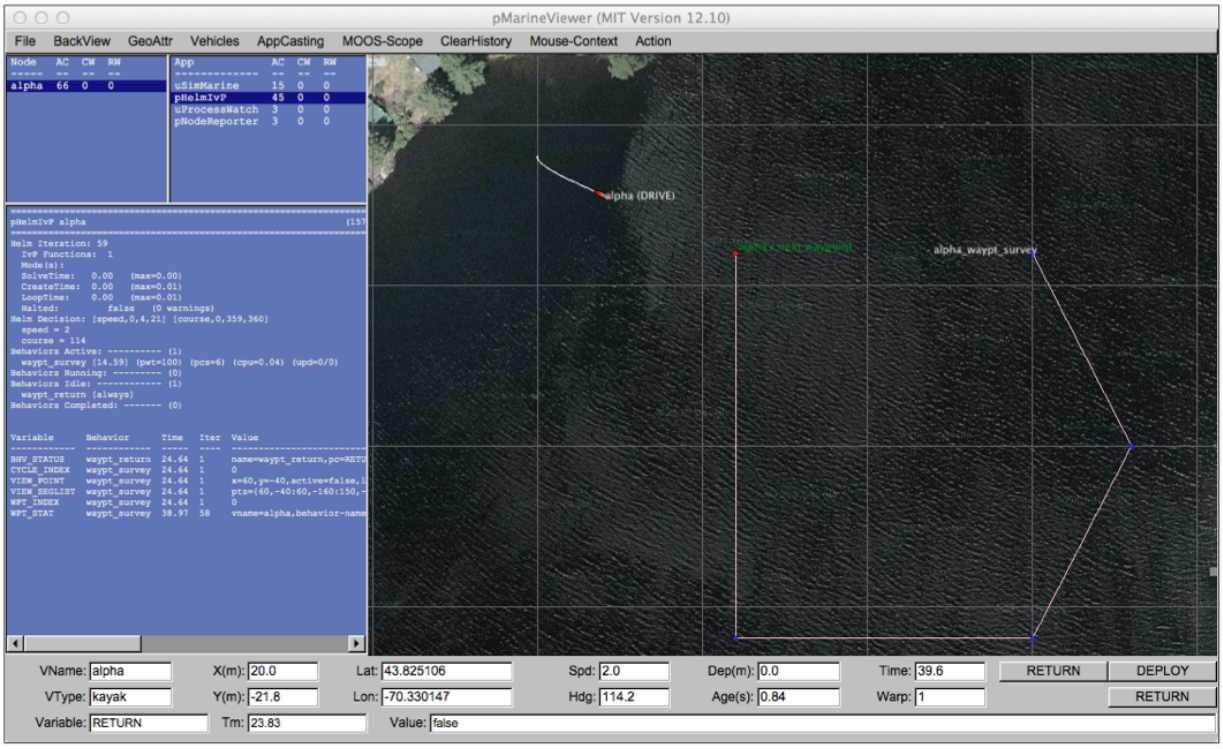


Figure 67: A screen shot of the `pMarineViewer` application running the alpha example mission. The position, heading, speed and other information related to the vehicle is reflected in the data fields at the bottom of the viewer.

Properties of the vehicle rendering such as the trail length, size, and color, and vehicle size and color, and pan and zoom can be adjusted dynamically in the GUI. They can also be set in the `pMarineViewer` MOOS configuration block. Both methods of tuning the rendering parameters are described later in this section. The individual fields of the data section are described below:

- `VName:` The name of the active vehicle associated with the data in the other GUI data fields. The

active vehicle is typically indicated also by changing to the color red on the geo display.

- `VType:` The platform type, e.g., AUV, Glider, Kayak, Ship or Unknown.
- `X(m):` The x (horizontal) position of the active vehicle given in meters in the local coordinate system.
- `Y(m):` The y (vertical) position of the active vehicle given in meters in the local coordinate system.
- `Lat:` The latitude (vertical) position of the active vehicle given in decimal latitude coordinates.
- `Lon:` The longitude (horizontal) position of the active vehicle given in decimal longitude coordinates.
- `Spd:` The speed of the active vehicle given in meters per second.
- `Hdg:` The heading of the active vehicle given in degrees $(0 - 359.99)$.
- `Dep(m):` The depth of the active vehicle given in meters.
- `Age(s):` The elapsed time in seconds since the last received node report for the active vehicle.
- `Time:` Time in seconds since the pMarineViewer process launched.
- `Warp:` The MOOS Time-Warp value. Simulations may run faster than real-time by this warp factor. MOOSTimeWarp is set as a global configuration parameter in the `.moos` file.
- `Range:` The range (in meters) of the active vehicle to a reference point. By default, this point is the datum, or the (0,0) point in local coordinates. The reference point may also be set to another particular vehicle. See Section 13.9 on the `ReferencePoint` pull-down menu.
- `Bearing:` The bearing (in degrees) of the active vehicle to a reference point. By default, this point is the datum, or the (0,0) point in local coordinates. The reference point may also be set to another particular vehicle. See Section 13.9 on the `ReferencePoint` pull-down menu.

The age of the node report is likely to remain zero in simulation as shown in the figure, but when operating on the water, monitoring the node report age field can be the first indicator when a vehicle has failed or lost communications. Or it can act as an indicator of communications link quality.

The lower three fields of the window are used for scoping on a single MOOS variable. See Section 13.6 for information on how to configure the `pMarineViewer` to scope on any number of MOOS variables and select a single variable via an optional pull-down menu. The scope fields are:

- `Variable:` The variable name of the MOOS variable currently being scoped, or `"n/a"` if no scope variables are configured.
- `Time:` The variable name of the MOOS variable currently being scoped, or `"n/a"` if no scope variables are configured.
- `Value:` The actual current value for the presently scoped variable.

13.1.3 The AppCasting, FullScreen and Traditional Display Modes

As mentioned above, appcasting is new to release 12.11, `pMarineViewer` supports three display modes. The first mode is the *normal* mode familiar to pre-12.11 users of `pMarineViewer` as it was the only mode. A second mode, the *appcasting* mode, also shows the three appcasting panes shown above in Figure 67. The third mode is the *full-screen* mode which shows only the geo-display part to maximize viewing of the operation area. The modes may be toggled by single hot-key actions as shown in the figure.

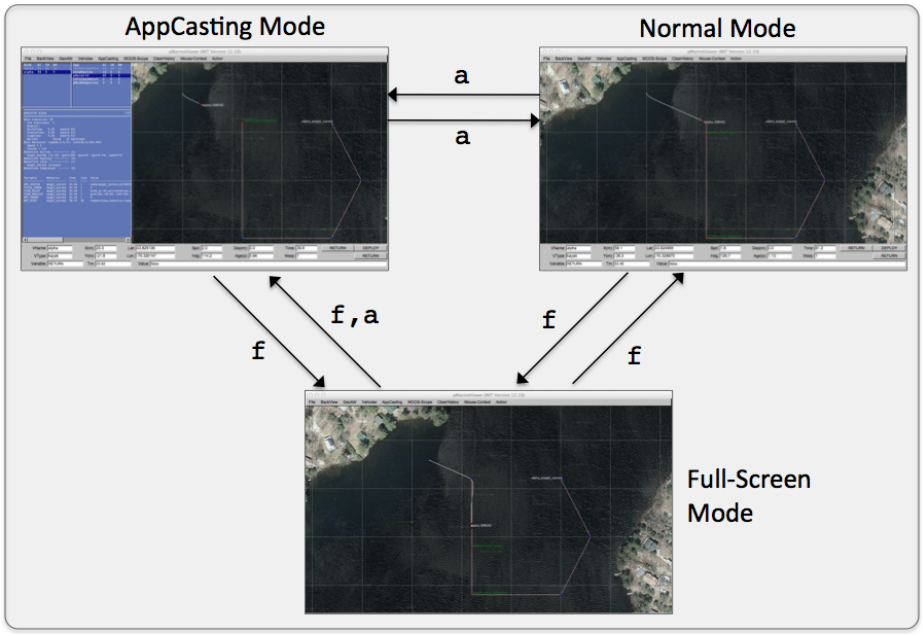


Figure 68: Three viewing modes are supported by `pMarineViewer`. The *normal* mode, the *appcasting* mode which renders appcast output from any connected vehicle, or the *full-screen* mode to maximize viewing of the operation area and vehicles. The modes may be toggled with the hot-keys shown. When typing `'f'` in the full-screen mode, the viewer will return to the mode prior to entering the full-screen mode. The modes may also be changed via pull-down menu items, or set to personal preferences in the `.moos` configuration block.

To launch a mission in the appcasting mode, set `appcast_viewable=true` in the `pMarineViewer` configuration block. To launch in the full-screen mode, set `full_screen=true` in the configuration block instead.

13.1.4 Run-Time and Mission Configuration

Nearly all `pMarineViewer` configuration parameters may be configured both at run-time, via pull-down menu selections, and prior to launch via configuration lines in the `pMarineViewer` configuration block of the `.moos` mission configuration file. To reduce the need to consult the documentation, the text of the pull-down menu selection is identical to the text of the parameter in the configuration file. Furthermore, most parameter selections are a choice from a fixed set of options. The present option for a parameter is typically indicated by a radio button in the pull-down menu.

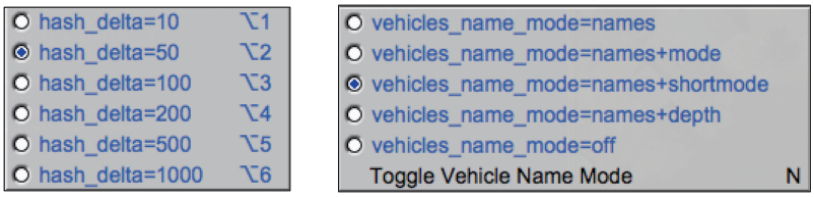


Figure 69: Most configuration parameters may be altered with pull-down menu selections. The radio-button shows the present parameter value and its neighbors show other legal settings. The text of the pull-down menu selection may be placed verbatim in the `.moos` configuration block to determine the setting upon the next mission launch. In general, menu items rendered in blue text are legally accepted parameters for placing in the `.moos` configuration block. Items in black are not.

Most parameter options have either a hot key associated with each option as shown in the left in Figure 69, or a hot key for toggling between options as on the right in the figure.

13.1.5 Command-and-Control

For the most part `pMarineViewer` is intended to be only a receiver of information from the vehicles and the environment. Adding command and control capability, e.g., widgets to re-deploy or manipulate vehicle missions, can be readily done, but make the tool more specialized, bloated and less relevant to a general set of users. However, `pMarineViewer` does have a few powerful extendible command and control capabilities under the hood. Each are simply ways to conveniently post to the MOOSDB, and come in three forms: (a) configurable pull-down menu actions, and (b) contextual mouse poking with embedded oparea information, and (c) configurable action buttons:

Configurable Pull-Down Menu Actions

The Action pull-down menu described in Section 13.7 provides a way to pre-define a set of MOOS postings, each selectable from the pull-down menu. For example, the alpha mission is configured with the below action:

```
action = RETURN = true
```

This post to the MOOSDB correlates to a behavior condition of the helm waypoint behavior with the return position. Actions may also be grouped into a single pull-down selection, discussed in Section 13.7.

Contextual Mouse Poking with Embedded OpArea Information

The mouse left and right buttons may be configured to make a post to the MOOSDB with value partly comprised of the point in the oparea under the mouse when clicked. For example, rather than commanding the vehicle to return to a pre-defined return position as the case above implies, the user may use this feature to command the vehicle to a point selected by the user at run time with a mouse click. The configuration might look like:

```
left_context[return] = RETURN_POINT = points = x=$(XPOS), y=$(YPOS)
left_context[return] = RETURN = true
```

This is discussed further in Section 13.8.

Action Button Configuration

Perhaps the most visible form of command and control is with the few action buttons configurable for on-screen use. For example, the DEPLOY and RETURN buttons in the lower right corner as in Figures 65, and 67. These buttons, for example, are configures as follows:

```
button_one = DEPLOY # DEPLOY=true
button_one = MOOS_MANUAL_OVERIDE=false # RETURN=false
button_two = RETURN # RETURN=true
```

The general syntax is:

```
button_one   = <label> # <MOOSVar>=<value> # <MOOSVar>=<value>  ...
button_two   = <label> # <MOOSVar>=<value> # <MOOSVar>=<value>  ...
button_three = <label> # <MOOSVar>=<value> # <MOOSVar>=<value>  ...
button_four  = <label> # <MOOSVar>=<value> # <MOOSVar>=<value>  ...
```

The left-hand side contains one of the four button keywords, e.g., button_one. The right-hand side consists of a '#'-separated list. Each component in this list is either a '='-separated variable-value pair, or otherwise it is interpreted as the button's label. The ordering does not matter and the '#'-separated list can be continued over multiple lines as in the simple example above.

The variable-value pair being poked on a button call will determine the variable type by the following rule of thumb. If the value is non-numerical, e.g., true, one, it is poked as a string. If it is numerical it is poked as a double value. If one really wants to poke a string of a numerical nature, the addition of quotes around the value will suffice to ensure it will be poked as a string.

13.2 The BackView Pull-Down Menu

The BackView pull-down menu deals mostly with panning, zooming and issues related to the rendering of the background on which vehicles and mission artifacts are rendered. The full menu is shown in Figure 70. Although panning and zooming is not something typically done via the pull-down menu, they are included in this menu primarily to remind the user of their hot-keys. The zooming commands affect the viewable area and apparent size of the objects. Zoom in with the 'i' or 'I' key, and zoom out with the 'o' or 'O' key. Return to the original zoom with ctrl+'z'.

13.2.1 Panning and Zooming

Panning is done with the keyboard arrow keys. Three rates of panning are supported. To pan in 20 meter increments, just use the arrow keys. To pan "slowly" in one meter increments, use the Alt + arrow keys. And to pan "very slowly", in increments of a tenth of a meter, use the Ctrl + arrow keys. The viewer supports two types of "convenience" panning. It will pan to put the active vehicle in the center of the screen with the 'C' key, and will pan to put the average of all vehicle positions at the center of the screen with the 'c' key. These are part of the 'Vehicles' pull-down menu discussed in Section 13.4.

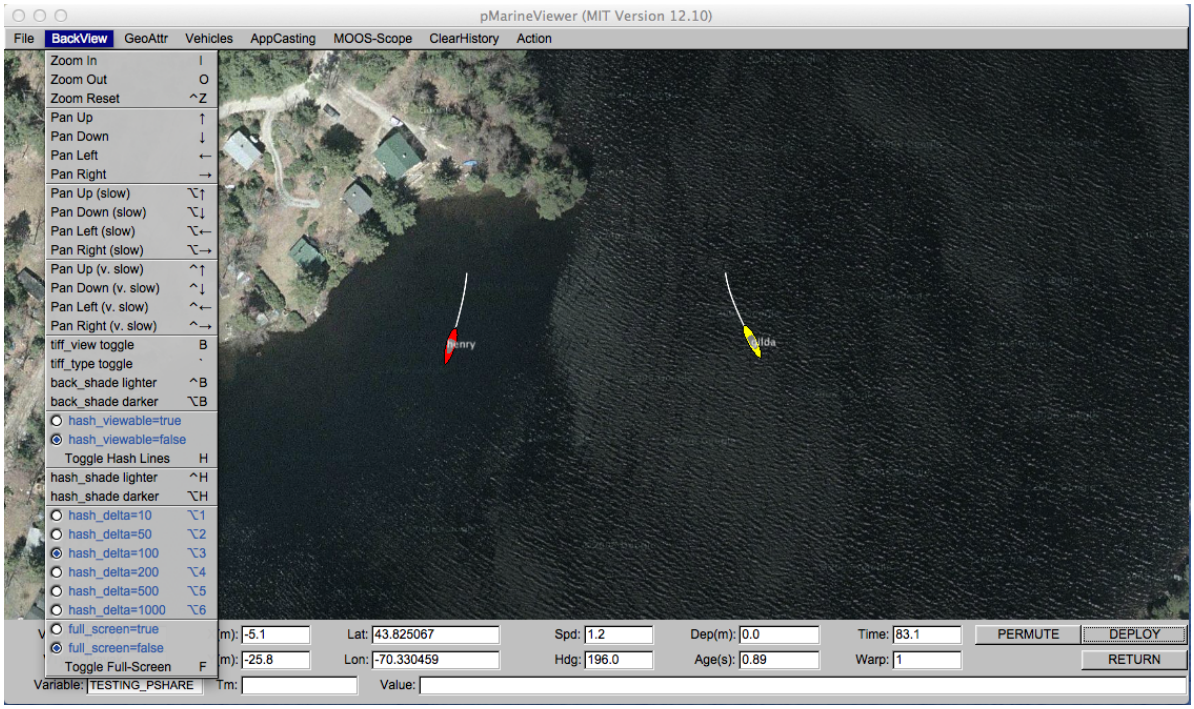


Figure 70: **The BackView menu:** This pull-down menu lists the options, with hot-keys, for affecting rendering aspects of the geo-display background.

13.2.2 Background Images

The background can be in one of two modes; either displaying a gray-scale background, or displaying a geo image read in as a texture into OpenGL from an image file. The default is the geo display mode if provided on start up, or the grey-scale mode if no image is provided. The mode can be toggled by typing the 'b' or 'B' key. The geo-display mode can have two sub-modes if two image files are provided on start-up. This is useful if the user has access to a satellite image *and* a map image for the same operation area. The two can be toggled by hitting the back tick key. When in the grey-scale mode, the background can be made lighter by hitting the ctrl+'b' key, and darker by hitting the alt+'b' key.

To use an image in the geo display, the input to `pMarineViewer` comes in two files, an image file in TIFF format, and an information text file correlating the image to the local coordinate system. The file names should be identical except for the suffix. For example `dabob_bay.tif` and `dabob_bay.info`. Only the `.tif` file is specified in the `pMarineViewer` configuration block of the MOOS file, and the application then looks for the corresponding `.info` file. The info file correlates the image to the local coordinate system and specifies the location of the local `(0,0)` point. An example is given in Listing 33.

Listing 33 - An example .info file associated with a background image.

```
1  // Lines may be in any order, blank lines are ok
2  // Comments begin with double slashes
3
4  datum_lat  = 47.731900
```

```
5  datum_lon  = -122.85000
6  lat_north  = 47.768868
7  lat_south  = 47.709761
8  lon_west   = -122.882080
9  lon_east   = -122.794189
```

All four latitude/longitude parameters are mandatory. The two datum lines indicate where $(0, 0)$ in local coordinates is in earth coordinates. However, the datum used by pMarineViewer is determined by the `LatOrigin` and `LongOrigin` parameters set globally in the MOOS configuration file. The datum lines in the above information file are used by applications other than `pMarineViewer` that are not configured from a MOOS configuration file. The `lat_north` parameters correlate the upper edge of the image with its latitude position. Likewise for the other three parameters and boundaries. Two image files may be specified in the `pMarineViewer` configuration block. This allows a map-like image and a satellite-like image to be used interchangeably during use. An example of this is shown in Figure 71 with two images of Dabob Bay in Washington State. Both image files where created from resources at www.maps.google.com.

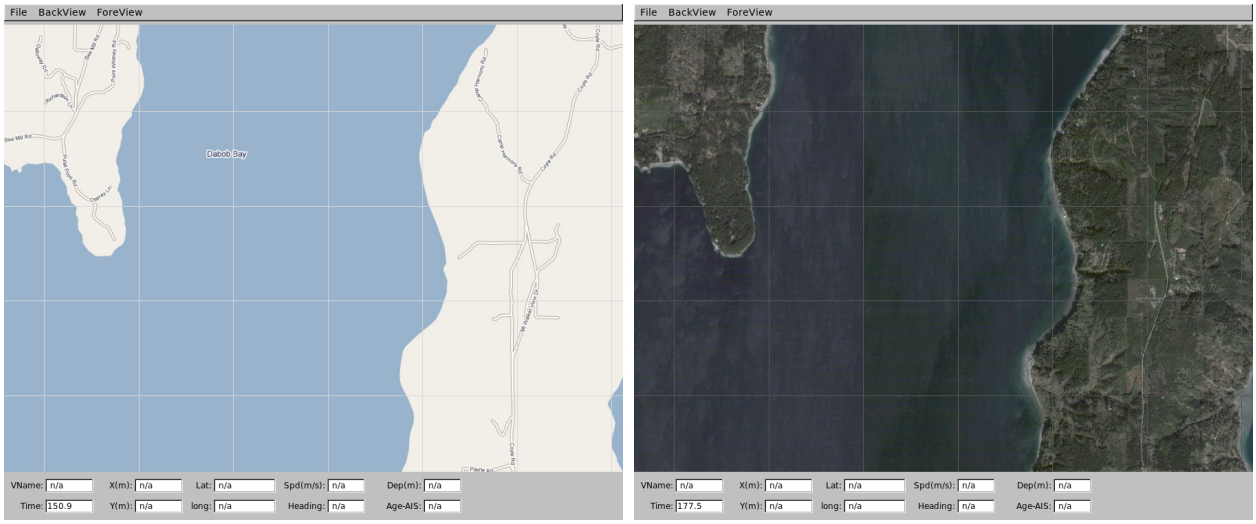


Figure 71: **Dual background geo images:** Two images loaded for use in the geo display mode of `pMarineViewer`. The user can toggle between both as desired during operation.

In the configuration block, the images can be specified by:

```
tiff_file   = dabob_bay_map.tif
tiff_file_b = dabob_bay_sat.tif
```

By default `pMarineViewer` will look for the files `Default.tif` and `DefaultB.tif` in the local directory unless alternatives are provided in the configuration block.

By default a copies of the background image and info files are *not* logged by `pLogger`. This may be changed by the setting the following parameter: `log_the_image = true`. This is only a request to `pLogger` in the form of the `PLOGGER_CMD` posting:

```
PLOGGER_CMD = COPY_FILE_REQUEST = /home/jake/images/lake_george.tif
PLOGGER_CMD = COPY_FILE_REQUEST = /home/jake/images/lake_george.info
```

The result should be that the files are included in the folder created by `pLogger` with the `._tif` and `._info` suffixes. These may then be used by post-mission analysis tools to re-convey the operation area.

13.2.3 Local Grid Hash Marks

Hash marks can be overlaid onto the background. By default this mode is off, but can be toggled with the 'h' or 'H' key, or set in the configuration file with the `hash_viewable` parameter. The hash marks are drawn in a grey-scale which can be made lighter by typing the ctrl+'h' key, and darker by typing the alt+'h' key, or set in the configuration file with the `hash_shade` parameter. The hash mark spacing may only be set to one of the values shown in the menu. If set to different value, the closest legal value will be chosen.

13.2.4 Full-Screen Mode

The viewer may be put into full-screen mode by toggling with the `'f'` key. This will result in the data fields at the bottom of the viewer being replace with a bit more viewing area for the geo display. As with all other blue items in this pull-down menu, the full-screen mode may be set in the MOOS configuration block with `full_screen=true`. The default if false. Full-screen mode is useful when running simulations while connected to a low-resolution projector for example.

13.3 The GeoAttributes Pull-Down Menu

The GeoAttributes pull-down menu allows the user to affect viewing properties of geometric objects capable of being rendered by the `pMarineViewer`. The viewer subscribes for and supports the following geometric objects, typically generated by the helm or other MOOS applications:

- Polygons
- SegLists
- Points
- Vectors
- Circles
- Markers
- RangePulses
- CommsPulses

The viewer will also render the following other geometric objects set either in the configuration file or interactively by the user:

- Datum
- OpArea
- DropPoints

The Datum is simply the point in local coordinates representing `(0,0)`. The pull-down menu allows the user to toggle off or on this rendering of the datum point as well as adjust its size and color. The OpArea is used to render the boundaries, if they exist, of an area of operation. DropPoints (described further in Section 13.3.5) are labeled points the user may drop on the viewing area for reference or mission planning

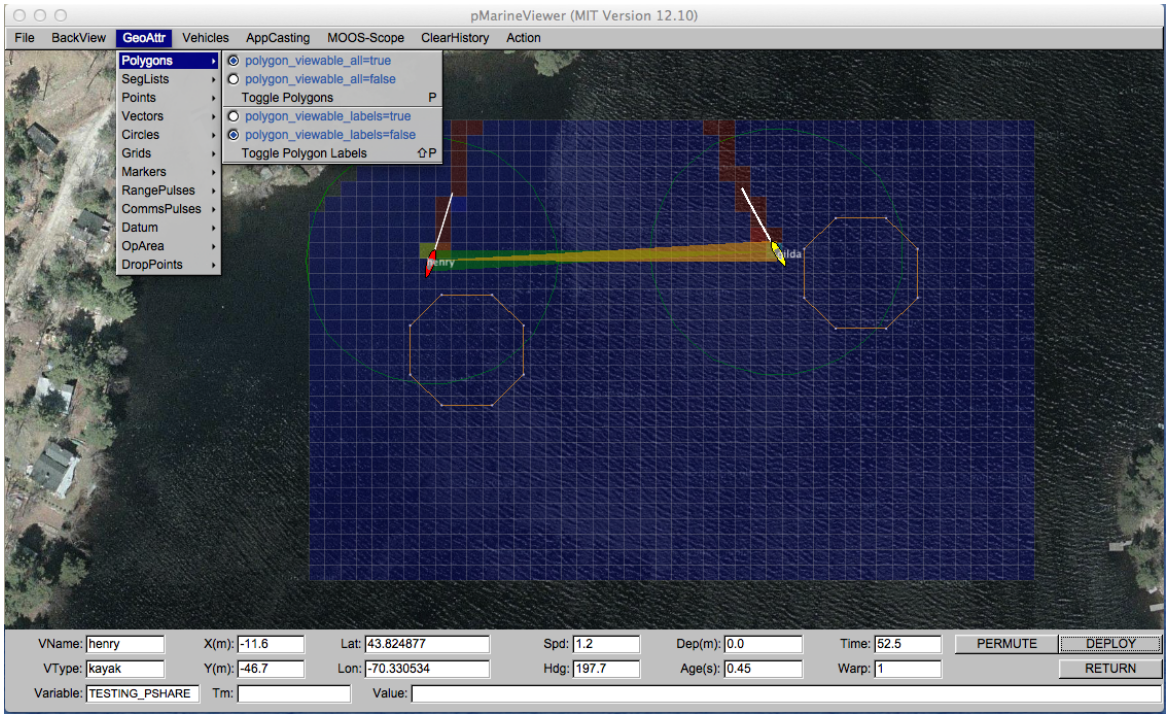


Figure 72: **The "GeoAttr" menu:** This pull-down menu lists the options and hot keys for affecting the rendering of geometric objects.

The possible parameters settings for rendering the geometric objects received by `pMarineViewer` via MOOS mail is provided in Section 13.10.2.

13.3.1 Polygons, SegLists, Points, Circles and Vectors

The five geometric objects, polygons, seglists, points, circles and vectors, provide a core rendering capability for applications (like the helm and its behaviors) to render visual artifacts related to the unfolding of a mission. For example, in Figure 65, a seglist is used to render the vehicle waypoints, and a labeled point is used to render the vehicles current next waypoint.

Objects are passed to `pMarineViewer` as strings via normal MOOS mail. An example is given below for the seglist shown in Figure 65. The string is a comma-separated list of variable=value pairs. Note the last pair is a label. Labels are used by all five object types to distinguish uniqueness.

```
VIEW_SEGLIST =  pts={60,-40:60,-160:150,-160:180,-100:150,-40},label=waypt_survey
```

Uniqueness is used to either overwrite or erase previously rendered object instances. For example the above seglist could be "moved" five meters south by posting an identical message with the same label and adjusted coordinates. The *source* of the object is also tracked by `pMarineVeiwer`. This is given by the MOOS community from which the message originated, and typically represents the vehicle's name. Thus the above seglist could also be "moved" if the posting originated from a second vehicle community, in the type of arrangement shown in Figure 66.

Parameters Common to Polygons, SegLists, Points, Circles and Vectors

Other optional parameters may be associated with an object to specify rendering preferences. They include:

- active
- msg
- vertex_size:
- vertex_color
- edge_size
- edge_color
- fill_color
- fill_transparency

For example, the VIEW_SEGLIST specification above may be augmented with the below string to specify edge and vertex size and color preferences:

```
edge_color=pink,vertex_color=blue,vertex_size=4,edge_size=1
```

The active parameter may be set to false to indicate that an object, previously received with the same label, should not be drawn by pMarineViewer. The msg parameter may be used to override the string rendered as the object's label. Since labels are used to uniquely identify an object, the msg parameter may be used to, for example, draw five points all with same rendered text. The other six parameters are self-explanatory and not necessarily relevant to all objects. For example, pMarineViewer will ignore an edge_size specification when drawing a point, and a fill_color will only be relevant for a polygon and a circle.

Serializing Geometric Objects for pMarineViewer Consumption

Geometric objects are only *consumed* by pMarineViewer, but it's worth discussing the issue of *generating* and serializing an object into a string. It is possible to simply post a string in the right format, as with:

```
string str = "x=5,y=25,label=home,vertex_size=3";  // Not recommended
m_Comms.Notify("VIEW_POINT", str);
```

It is highly recommended that this be left to the serialization function native to the C++ class.

```
#include "XYPoint.h"

XYPoint my_point(5, 25);                          // Recommended
my_point.set_label("home");
my_point.set_vertex_size(3);
string str = my_point.get_spec();
m_Comms.Notify("VIEW_POINT", str);
```

The latter code is less prone to user error, and is more likely to work in future code releases if the underlying formats need to be altered. (This is the idea behind Google Protocol Buffers, but here the geometric classes are implemented with various geometry function relations defined in addition to the serialization and de-serialization.) The full set of interface possibilities for creating and manipulating geometry objects is beyond the scope of the discussion here however.

13.3.2 Markers

A set of marker object types are defined for rendering characteristics of an operation area such as buoys, fixed sensors, hazards, or other things meaningful to a user. The six types of markers are shown in Figure 73. They are configured in the `pMarineViewer` configuration block of the MOOS file with the following format:

```
marker  = type=efield,x=100,y=20,label=alpha,color=red,width=4.5
marker  = type=square,lat=42.358,lon=-71.0874,color=blue,width=8
```

Each entry is a string of comma-separated pairs. The order is not significant. The only mandatory fields are for the marker type and position. The position can be given in local x-y coordinates or in earth coordinates. If both are given for some reason, the earth coordinates will take precedent. The `width` parameter is given in meters drawn to scale on the geo display. Shapes are roughly 10x10 meters by default. The GUI provides a hook to scale all markers globally with the `ALT-m` and `CTRL-m` hot keys and in the GeoAttributes pull-down menu.

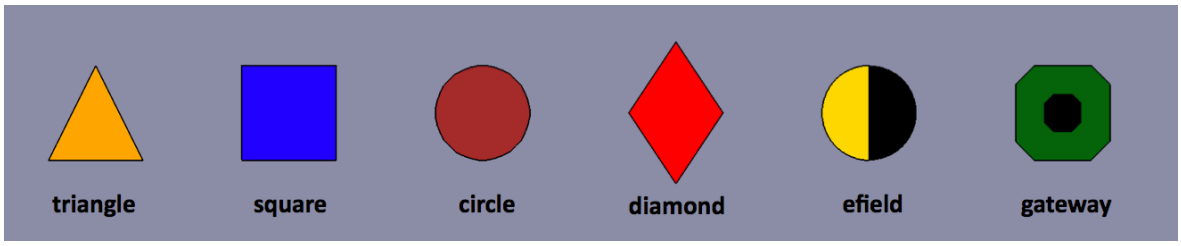


Figure 73: **Markers:** Types of markers known to the `pMarineViewer`.

The color parameter is optional and markers have the default colors shown in Figure 73. Any of the colors described in the Colors Appendix are fair game. The black part of the Gateway and Efield markers is immutable. The label field is optional and is by default the empty string. Note that if two markers of the same type have the same non-empty label, only the first marker will be acknowledged and rendered. Two markers of different types can have the same label.

In addition to declaring markers in the configuration file, markers can be received dynamically by `pMarineViewer` through the `VIEW_MARKER` MOOS variable, and thus can originate from any other process connected to the `MOOSDB`. The syntax is exactly the same, thus the above two markers could be dynamically received as:

```
VIEW_MARKER  = "type=efield,x=100,y=20,scale=4.3,label=alpha,color=red,width=4.5"
VIEW_MARKER  = "type=square,lat=42.358,lon=-71.0874,scale=2,color=blue,width=8"
```

The effect of a "moving" marker, or a marker that changes color, can be achieved by repeatedly publishing to the `VIEW_MARKER` variable with only the position or color changing while leaving the label and type the same. To dynamically alter the text rendered with a marker, the `msg=value` field may be used instead. When the message is non-empty, it will be rendered instead of the label text.

13.3.3 Comms Pulses

Comms pulse objects were designed to convey a passing of information from one node to another. At this writing, they are only used by the `uFldNodeComms` application, but from the perspective of `pMarineViewer` it does not matter the origin. The MOOS variable is `VIEW_COMMS_PULSE`. They look something like that shown in Figure 74. There are two pulses shown in this figure. In this case they were posted by `uFldNodeComms` to indicate that the two vehicles are receiving each other's node reports.

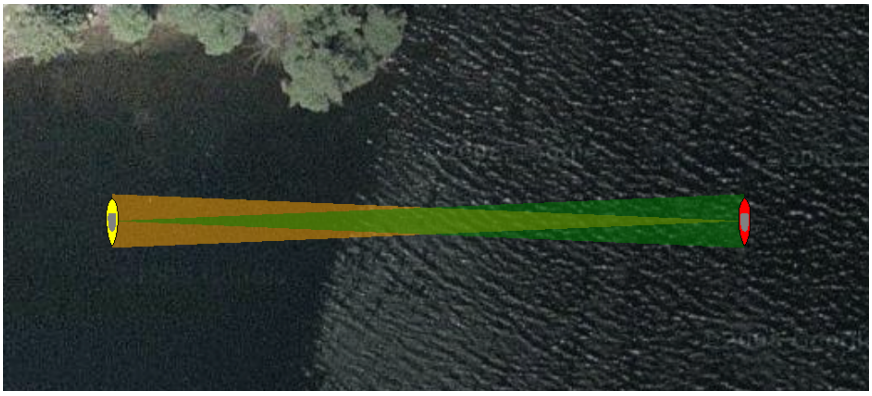


Figure 74: **Comms Pulses:** A comms pulse directionally renders communication between vehicles. Here each vehicle is communicating with the other, and two different colored pulses are rendered.

The term "pulse" is used because the object has a duration (by default three seconds), after which it will no longer be rendered by `pMarineViewer`. The pulse will fade (become more transparent) linearly with time as it approaches its expiration. If a subsequent comms pulse is received with an identical label before the first pulse times out, the second pulse will replace the first, in the style of other geometric objects discussed previously. Although serializing and de-serializing comms pulse messages is outside the scope of this discussion, it it worth examining an example comms pulse message:

```
VIEW_COMMS_PULSE = sx=91,sy=29,tx=6.7,ty=1.4,beam_width=7,duration=10,fill=0.35,
                   label=GILDA2HENRY_MSG,fill_color=white,time=1350201497.27
```

As with the object types discussed previously, the construction of the above type messages should be handled by the `XYCommsPulse` class along the line of something like:

```
#include "XYCommsPulse.h"

XYCommsPulse my_pulse(91, 29, 6.7, 1.4);
my_pulse.set_label("GILDA2HENRY_MSG");
my_pulse.set_duration(10);
my_pules.set_beam_width(7);
my_pules.set_fill(0.35);
my_pulse.set_color("fill", "white");
string str = my_pulse.get_spec();
m_Comms.Notify("VIEW_COMMS_PULSE", str);
```

The white comms pulse shown in Figure 75 indicates that a message has been sent from one vehicle to the other. The fat end of the pulse indicates the receiving vehicle. The color scheme is not a convention of `pMarineViewer`, but rather a convention of the `uFldNodeComms` application which generated the object in this case. A white pulse is typically rendered long enough to allow the user to visually register the information. It also typically does not move with the vehicle, to convey to the user the vehicle positions at the time of the communication.

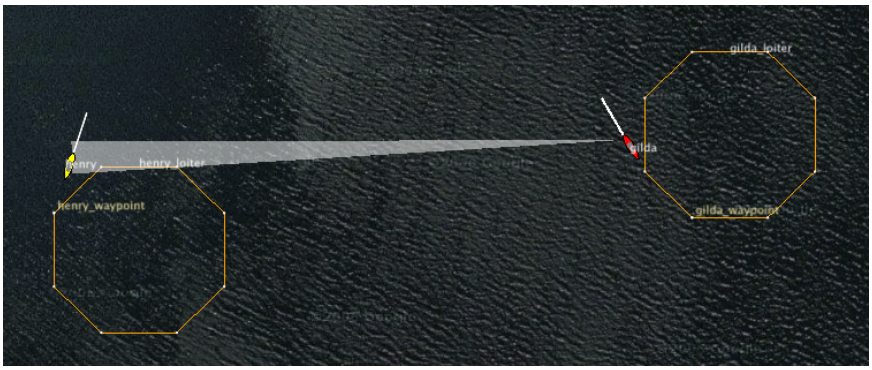


Figure 75: **Comms Pulses for Messaging:** In this figure the white comms pulse indicates that a message is being sent from one vehicle to another, via `uFldNodeComms`.

The rendering of comms pulses may be toggled on or off in `pMarineViewer` via a selection in the GeoAttr pull-down menu, or via the `'@'` hot key. It is not possible in pMarineViewer to show just the white comms pulses, and hide the colored node report comms pulses, or vice versa. It is possible however in the `uFldNodeComms` configuration to shut off the node report pulses with `view_node_report_pulses=false`.

13.3.4 Range Pulses

Range pulse objects were designed to convey a passing of information or sensor energy from one node to any other node in the vicinity, up to a certain range. At this writing they are only used by the `uFldContactRangeSensor` and `uFldBeaconRangeSensor` applications, but from the perspective of `pMarineViewer` it does not matter the origin. The MOOS variable is `VIEW_RANGE_PULSE`. They look something like that shown in Figure 76. Here the pulse is shown over three successive times.

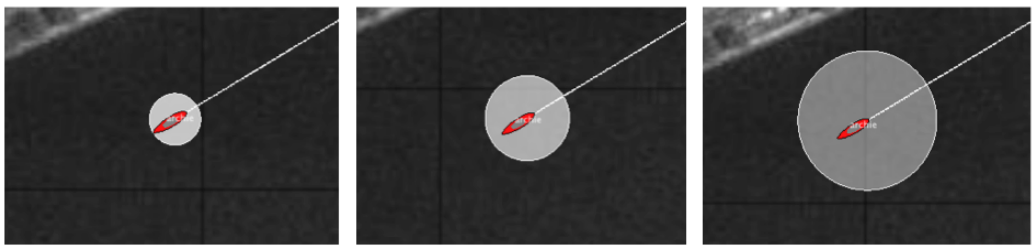


Figure 76: **Comms Pulses:** A comms pulse directionally renders communication between vehicles. Here each vehicle is communicating with the other, and two different colored pulses are rendered.

The term "pulse" is used because the object has a duration (by default 15 seconds), after which

it will no longer be rendered by pMarineViewer. The pulse will grow in size and fade (become more transparent) linearly with time as it approaches its expiration. If a subsequent range pulse is received with an identical label before the first pulse times out, the second pulse will replace the first, in the style of other geometric objects discussed previously. Although serializing and de-serializing range pulse messages is outside the scope of this discussion, it it worth examining an example range pulse message:

```
VIEW_RANGE_PULSE = x=99.2,y=68.9,radius=50,duration=6,fill=0.9,label=archie_ping,
                   edge_color=white,fill_color=white,time=2700438154.35,edge_size=1
```

As with the object types discussed previously, the construction of the above type messages should be handled by the XYRangePulse class along the line of something like:

```
#include "XYRangePulse.h"

XYRangePulse my_pulse(99.2, 68.9);
my_pulse.set_label("archie_ping");
my_pulse.set_duration(6);
my_pules.set_edge_size(1);
my_pules.set_radius(50);
my_pules.set_fill(0.9);
my_pulse.set_color("edge", "white");
my_pulse.set_color("fill", "white");
string str = my_pulse.get_spec();
m_Comms.Notify("VIEW_RANGE_PULSE", str);
```

13.3.5 Drop Points

A user may be interested in determining the coordinates of a point in the geo portion of the pMarineViewer window. The mouse may be moved over the window and when holding the SHIFT key, the point under the mouse will indicate the coordinates in the local grid. When holding the CTRL key, the point under the coordinates are shown in lat/lon coordinates. The coordinates are updated as the mouse moves and disappear thereafter or when the SHIFT or CTRL keys are release. Drop points may be left on the screen by hitting the left mouse button at any time. The point with coordinates will remain rendered until cleared or toggled off. Each click leaves a new point, as shown in Figure 77.

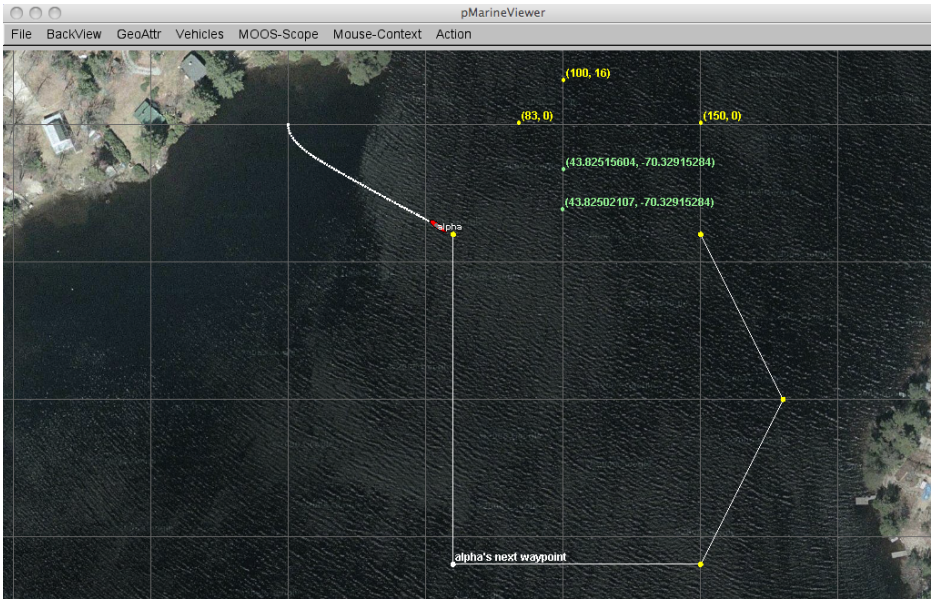


Figure 77: **Drop points:** A user may leave drop points with coordinates on the geo portion of the pMarineViewer window. The points may be rendered in local coordinates or in lat/lon coordinates. The points are added by clicking the left mouse button while holding the `SHIFT` key or `CTRL` key. The rendering of points may be toggled on/off, cleared in their entirety, or reduced by popping the last dropped point.

Parameters regarding drop points are accessible from the `GeoAttr` pull-down menu. The rendering of drop points may be toggled on/off by hitting the 'r' key. Drop points may also be shut off in the mission configuration file with `drop_point_viewable_all=false`. The set of drop points may be cleared in its entirety via the pull-down menu. Or the most recently dropped point may be removed by typing the `CTRL-r` key. The pull-down menu may also be used to change the rendering of coordinates from `"as-dropped"` where some points are in local coordinates and others in lat/lon coordinates, to `"local-grid"` where all coordinates are rendered in the local grid, or `"lat-lon"` where all coordinates are rendered in the lat/lon format. By default the mode is `"as-dropped"`. The startup default mode may be changed with `drop_point_coords=local-grid` for example in the mission file.

13.4 The Vehicles Pull-Down Menu

The *Vehicles* pull-down menu deals with rendering properties of vehicles, vehicle labels, and vehicle trails. The options are shown in Figure 78. The very first option is to turn on or off the rendering of all vehicles. The can be done at run time via the menu selection, or toggled with the `Ctrl-'a'` hot key. Like all blue options in this menu, the text in the menu item may be placed verbatim in the mission configuration file to reflect the user's startup preferences.

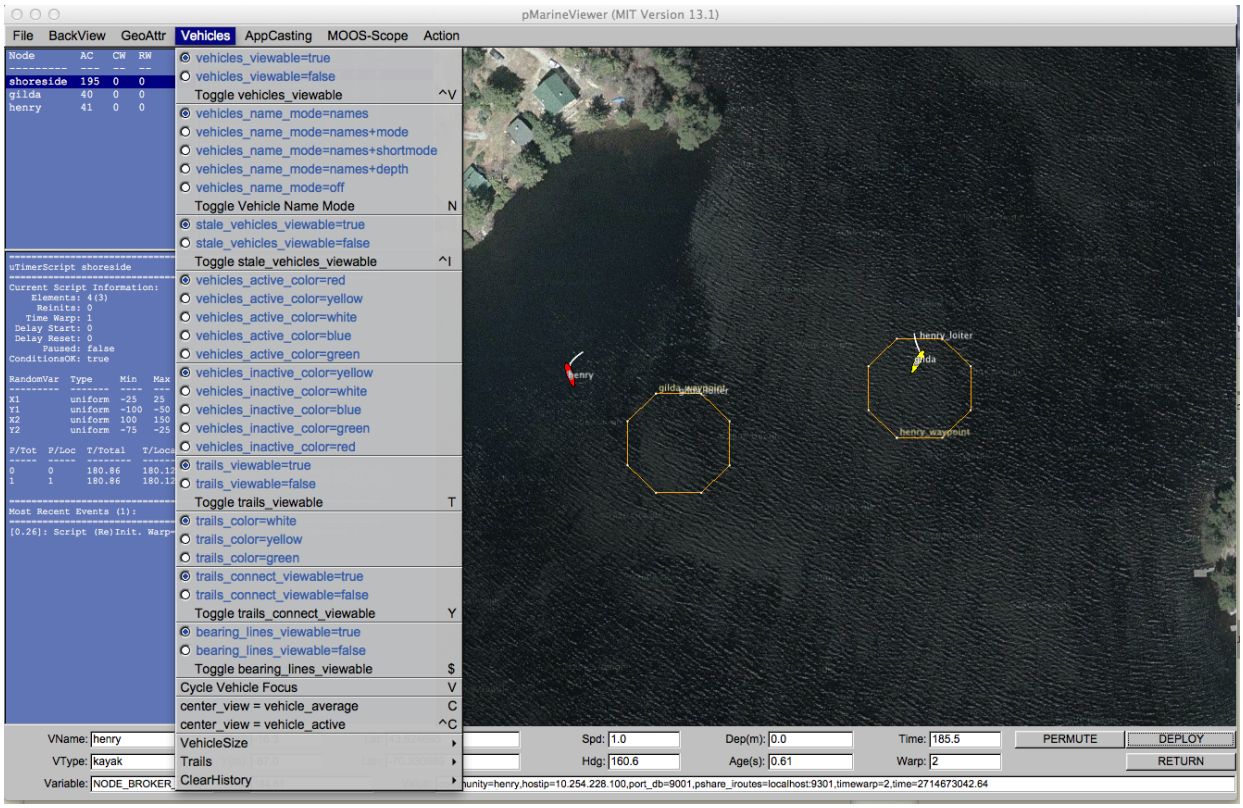


Figure 78: **The Vehicles menu:** This pull-down menu lists the options, with hot-keys, for affecting rendering vehicles and vehicle track history.

13.4.1 The Vehicle Name Mode

Each vehicle rendered in the viewer has an optional label rendered with it. This label may be rendered in one of five modes:

- *names*: Just the vehicle name is rendered.
- *names+mode*: The vehicle name and the full helm mode is rendered.
- *names+shortmode*: The vehicle name and the short helm mode is rendered.
- *names+depth*: The vehicle name and its current depth are rendered.
- *off*: No label is rendered.

The default is *names+shortmode*. The *names*, *off* and *depth* modes are self explanatory. The *names+mode* and *names+shortmode* involve information typically provided in vehicle node reports

about the state of the IvP helm. The helm uses hierarchical mode declarations as a way of configuring behaviors for missions. The helm mode for example be described with string looking something like `"MODE@ACTIVE:LOITERING"`. In `pMarineViewer` the text next to the vehicle would be either this whole string if configured with the *names+mode* setting, or just `"LOITERING"` if configured with the *names+shortmode* setting.

The color of the rendered text may be changed from the default of white to any color in Appendix C with the `vehicles_name_color` configuration parameter.

13.4.2 Dealing with Stale Vehicles

A stale vehicle is one who has not been heard from for a long time, perhaps because the vehicle is disabled, out of range, or recovered from the field. These vehicles can be a distraction. Their history may be outright cleared as described in Section 13.4.6, but this requires action by the user or a posting to the MOOSDB.

Stale vehicles are also automatically dealt with by `pMarineViewer` in another way. After some number of seconds (60 by default), the vehicle label indicates the staleness. The label may look something like `"henry (Stale Report: 231)"` where the number indicates the number of seconds since the last node report received for this vehicle. After another period of time (300 by default), the vehicle may no longer rendered. Note this does not remove the vehicle trail or any other geometric objects posted by the vehicle. To clear these, explicit action must be taken as described in Section 13.4.6. Often just clearing the vehicle itself is sufficient to reduce the distraction.

A few features of this policy are configurable at run time and through the mission configuration file. The policy of showing stale vehicles is false by default but may be toggled with the `Ctrl-'i'` key. It may also be set to be true with the `stale_vehicles_viewable=true` parameter. The duration of time after which a vehicle is reported as stale may be changed from its default of 60 seconds with the `stale_report_thresh` parameter. The duration of time after which a vehicle is no longer drawn may be changed from its default of 300 seconds with the `stale_nodraw_thresh` parameter.

13.4.3 Supported Vehicle Shapes

The shape rendered for a particular vehicle depends on the *type* of vehicle indicated in the node report received in `pMarineViewer`. There are four types that are currently handled, an AUV shape, a glider shape, a kayak shape, and a ship shape, shown in Figure 79.

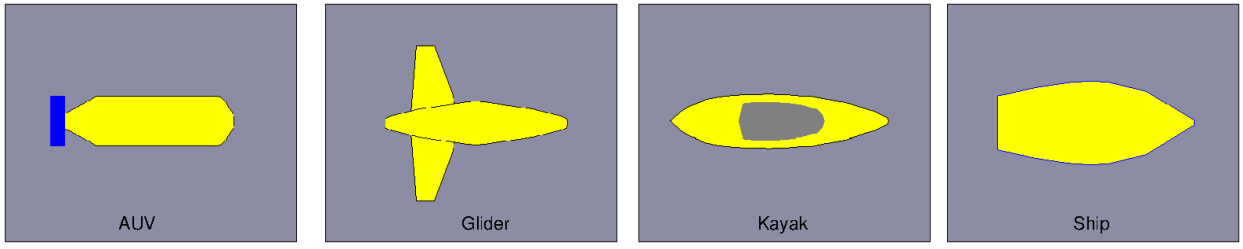


Figure 79: **Vehicles:** Types of vehicle shapes supported by `pMarineViewer`.

The default shape for an unknown vehicle type is currently set to be the shape "ship".

13.4.4 Vehicle Colors

Vehicles are rendered in one of two colors, the *active vehicle color* and the *inactive vehicle color*. The active vehicle is the one who's data is being rendered in the data fields at the bottom of the `pMarineViewer` window, and who's name is in the `VName:` field. The active vehicle may be changed by selecting `"Cycle Vehicle Focus"` from the Vehicles pull-down menu, or toggling through with the `'v'` key. The default color for the active vehicle is red, and the default for the inactive vehicle is yellow. These can be changed via the pull-down menu, or with the following parameters in the configuration file:

```
vehicles_active_color   = <color>    // default is red
vehicles_inactive_color = <color>    // default is yellow
```

The parameters and colors are case insensitive. All colors of the form described in Appendix C are acceptable.

13.4.5 Centering the Image According to Vehicle Positions

The `center_view` menu items alters the center of the view screen to be panned to either the position of the active vehicle, or the position representing the average of all vehicle positions. Once the user has selected this, this mode remains *sticky*, that is the viewer will automatically pan as new vehicle information arrives such that the view center remains with the active vehicle or the vehicle average position. As soon as the user pans manually (with the arrow keys), the viewer breaks from trying to update the view position in relation to received vehicle position information. The rendering of the vehicles can made larger with the '+' key, and smaller with the '-' key, as part of the `VehicleSize` pull-down menu as shown. The size change is applied to all vehicles equally as a scalar multiplier. Currently there is no capability to set the vehicle size individually, or to set the size automatically to scale.

13.4.6 Vehicle Trails

Vehicle trail (track history) rendering can be toggled off and on with the `'t'` key. The default is on. The startup default setting may be changed to off in the mission configuration file with `trails_viewable=false`.

Trail Color and Point Size

The trail color by default is white. A few other colors are available in the Vehicles pull-down menu. A color may also be chosen in the mission configuration file with `trail_color=<color>` using any color listed in Appendix C. The trail point size may range from $[1, 10]$. The default setting is 2. The size may be changed at runtime by the Vehicles/Trails pull-down menu, or with the `'{'` and `'}'` hot keys. The startup trail size may also be set in the mission configuration file with `trails_point_size=<int>` parameter.

Trail Length and Connectivity

Trails have a fixed-length history by default of 100 points. This may be changed via the Vehicles/-Trails pull-down menu, or with the hot keys `'('` and `')'`. The startup default length may also be

set in the mission configuration file with trails_length=<int> with values in range of $[0, 10000]$.

Individual trail points can be rendered with a line connecting each point, or by just showing the points. When the node report stream is flowing quickly, typically the user doesn't need or want to connect the points. When the viewer is accepting input from an AUV with perhaps a minute or longer delay in between reports, the connecting of points is helpful. This setting can be toggled with the 'y' or key, with the default being off. The startup default may be set to on with the mission file paramter trails_connect_viewable=true.

Resetting or Clearing the Trails

A vehicle's history sometimes needs to be cleared, for example when a vehicle has not been heard from in a long time, or has been recovered. Its trails and other geometric objects posted to the viewer can become a distraction. This may be done in a couple ways. First via the Action pull-down menu, the last menu item allows the user to clear the history of all vehicles or a selected vehicle. The Ctrl-9 hot key can be used to clear all vehicle histories. A select vehicle history may also be cleared by posting to the MOOS variable TRAIL_RESET with the name of the vehicle.

13.5 The AppCast Pull-Down Menu

With the addition of appcasting to MOOS, pMarineViewer has been augmented to serve as an appcast viewer (along with the other appcast viewer tools, uMAC, and uMACView). The motivation for appcasting and how to build appcast enabled MOOS applications are discussed elsewhere in Section **??**. The focus here is on the AppCast menu items and their effect on rendering to the user.

13.5.1 Turning On and Off AppCast Viewing

The AppCast pull-down menu, shown in Figure 80 allows adjustments to be made to the appcast rendering. The very first set of menu options allows the user to control whether the set of appcasting panes is viewed or not. The first two menu items allow the explicit on or off selection and also indicate the mission configuration parameter to turn appcast panes off by default, appcast_viewable=false. The third menu option allows the user to toggle the present setting and show that the 'a' key can be alternately used as shortcut for toggling.

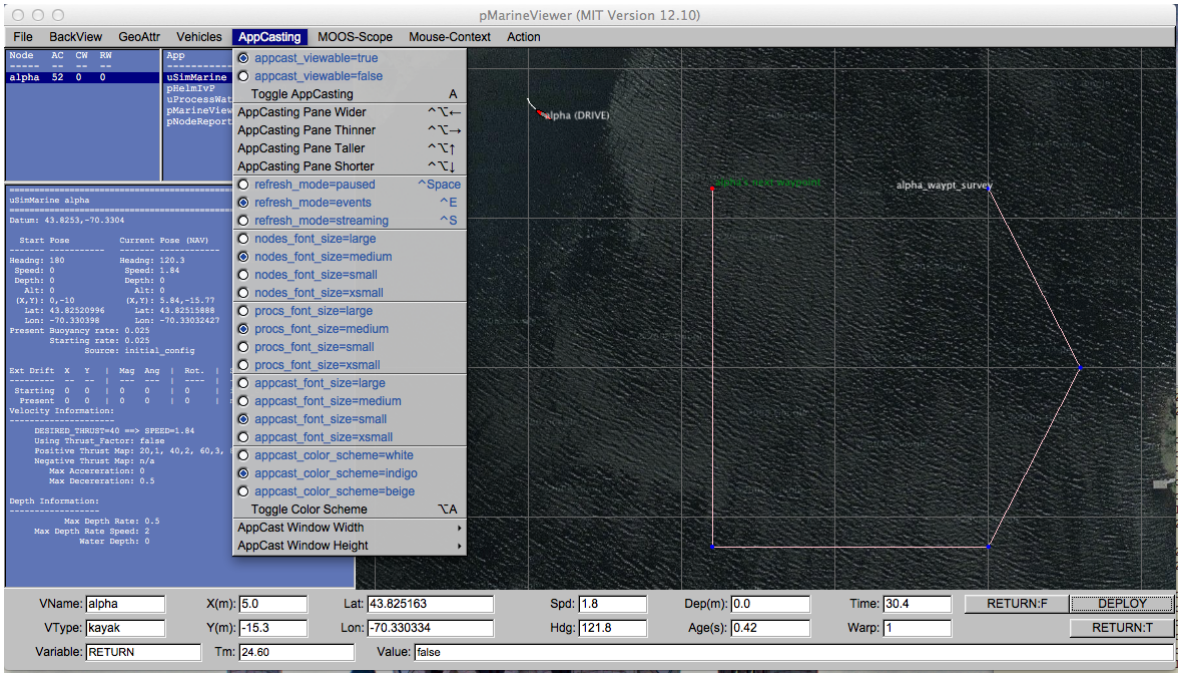


Figure 80: **The AppCast menu:** This pull-down menu lists the options, with hot-keys, for affecting rendering aspects of the appcast panels, and policy for soliciting appcasts from known vehicles and applications.

13.5.2 Adjusting the AppCast Viewing Panes Height and Width

The next set of menu items allow the relative size of the appcasting panes to be adjusted. The width of the three panes may be increased or decreased with the left and right arrow keys, and the height of the lower appcasting window relative to the two upper windows may be adjusted with the up and down arrow keys. In both cases, along with the arrow keys, the user must also hold down the `Ctrl` and `Alt` keys. Alternatively the `Ctrl` and `Shift` keys may be uses. Both modes are supported since user key bindings vary between users. The `Alt` + arrow keys are common in Ubuntu for switching work spaces for example.

The appcast pane extents may also be set to the user's liking in the mission configuration file with the parameters `appcast_width` and `appcast_height`. The allowable range of values for each may be seen by pulling down the "AppCast Window Width" and "AppCast Window Height" sub-menus of the AppCast pull-down menu.

13.5.3 Adjusting the AppCast Refresh Mode

The appcast *refresh mode* refers to the policy of sending appcast requests to known vehicles and applications. This is discussed more fully in Section **??**, but summarized here. Appcasting apps are implemented to be lazy with respect to generating appcasts - they will not generate them unless asked. And even when asked, the request comes with an expiration after which, if no new request has been received, the application returns to the lazy mode of producing no appcasts. So, for `pMarineViewer` to function as an appcast viewer, under the hood it must be also generating appcast requests (`APPCAST_REQ` postings) to the MOOSDB. The refresh mode refers to this under-the-hood

policy.

In the *paused* refresh mode, `pMarineViewer` is not generating any appcast requests at all. This is not the default and typically not very helpful, but it may be useful when the viewer is situated in the field with only a low-bandwidth connection to remote vehicles. The refresh mode may be set to *paused* via the pull-down menu selection, with the `CTRL+Spacebar` hot key, or set in the mission configuration file with `refresh_mode=paused`.

In the *events* refresh mode, the default mode, `pMarineViewer` is generating appcast requests only to the selected vehicle and the selected MOOS application. Even this is only partly true. In fact it is generating another kind of appcast request to all vehicles and apps, but this kind of request comes with the caveat that an app is only to generate an appcast report if a new run warning has been detected. Otherwise these apps remain lazy. In this mode you should expect to see regular appcasts received for the selected app, and updates for the other apps only if something worthy of a run warning has occurred. You can confirm this for yourself by looking at the counter reflecting the number of appcasts received for any application. This counter is under the `AC` column in the upper panes. The refresh mode may be set to *events* via the pull-down menu selection, with the `CTRL+'e'` hot key, or set in the mission configuration file with `refresh_mode=events`. The latter would be redundant however since this is the default mode.

In the *streaming* refresh mode, `pMarineViewer` is generating appcast requests to all vehicles and all apps to generate appcasts all the time. This mode is a bandwidth hog, but it may be useful at times, especially to debug why a particular application is silent. If it is not generating and appcast in this mode, then something may indeed be wrong. The refresh mode may be set to *streaming* via the pull-down menu selection, with the `CTRL+'s'` hot key, or set in the mission configuration file with `refresh_mode=streaming`.

13.5.4 Adjusting the AppCast Fonts

The font size of the text in the appcasting panes may be adjusted. There are three panes:

- *Nodes Pane*: The upper left pane shows the list of nodes (typically synonymous with vehicles), presently known to the viewer.
- *Procs Pane*: The upper right pane show the list of apps, for the chosen node, presently known to the viewer.
- *AppCast Pane*: The bottom pane shows the contents of the presently selected appcast report.

For each pane the possible font settings are `large`, `medium`, `small`, and `xsmall`. The default for the upper panes is `medium`, and the default for the appcast pane is `small`. Font sizes may be changed via the pull-down menu or set to the user's liking in the mission configuration file with `nodes_font_size`, `procs_font_size`, and `appcast_font_size` parameters.

13.5.5 Adjusting the AppCast Color Scheme

A few different color schemes governing the three appcast panes are available. The default color scheme is `"indigo"`, reflected in the Figure 80 for example. The other two color schemes are `"white"` and `"beige"`. The color scheme may be changed via the pull-down menu, toggled with the `ALT+'a'` hot key, or set to the user's liking in the mission configuration file with `appcast_color_scheme` parameter.

13.6 The MOOS-Scope Pull-Down Menu

The MOOS-Scope pull-down menu allows the user to configure `pMarineViewer` to scope on one or more variables in the MOOSDB. The viewer allows visual scoping on only a single variable at a time, but the user can select different variables via the pull-down menu, or toggle between the current and previous variable with the `'/'` key, or cycle between all registered variables with the `CTRL+'/'` key. The scope fields are on the bottom of the viewer as shown in Figure 81.

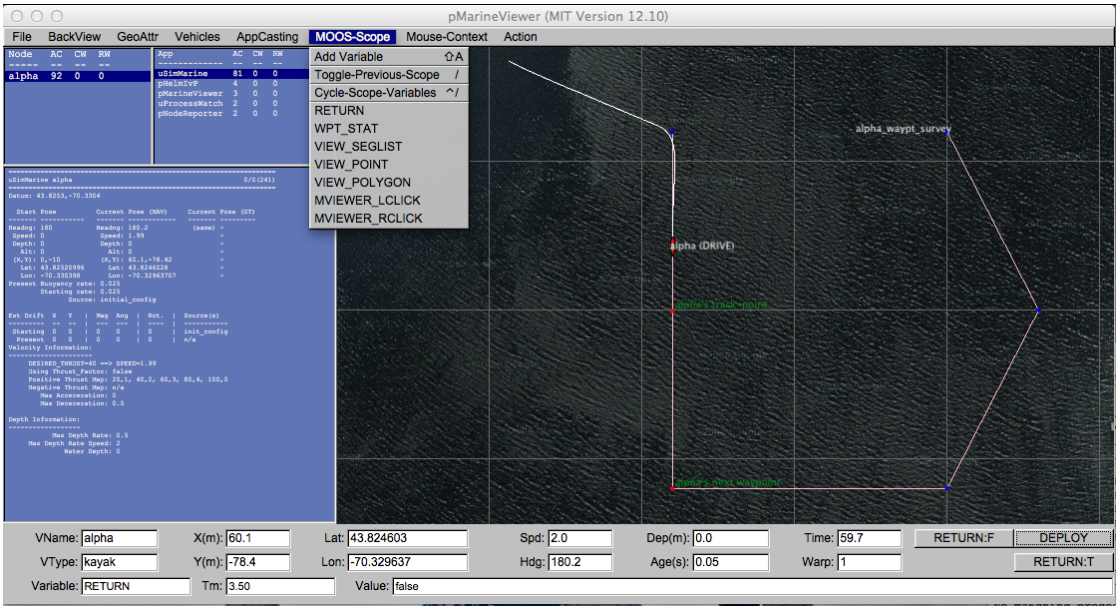


Figure 81: **The Scope Menu:** This pull-down menu allows the user to adjust which pre-configured MOOS variable is to be scoped, or to add a new variable to the scope list.

The three fields show (a) the variable name, (b) the last time is was updated, and (c) the current value of the variable. Configuration of the menu is done in the MOOS configuration block with entries similar to the following (which correlate to the particular items in the pull-down menu in Figure 81):

```
scope = RETURN, WPT_STAT, VIEW_SEGLIST, VIEW_POINT, VIEW_POLYGON
scope = MVIEWER_LCLICK, MVIEWER_RCLICK
```

The keyword `scope` is not case sensitive, but the MOOS variables are. If no entries are provided in the MOOS configuration block, the pull-down menu contains a single item, the `"Add Variable"` item. By selecting this, the user will be prompted to add a new MOOS variable to the scope list. This variable will then immediately become the actively scoped variable, and is added to the pull-down menu.

13.7 The Action Pull-Down Menu

The Action pull-down menu allows the user to invoke pre-define pokes to the MOOSDB (the MOOSDB to which the pMarineViewer is connected). While hooks for a limited number of pokes

are available by configuring on-screen buttons (Section 13.1.5), the number of buttons is limited to four. The "Action" pull-down menu allows for as many entries as will reasonably be shown on the screen. Each action, or poke, is given by a variable-value pair, and an optional grouping key. Configuration is done in the MOOS configuration block with entries of the following form:

```
action = menu_key=<key> # <MOOSVar>=<value> # <MOOSVar>=<value> # ...
```

If no such entries are provided, this pull-down menu will not appear. The fields to the right of the `action` are separated by the `'#'` character for convenience to allow several entries on one line. If one wants to use the `'#'` character in one of the variable values, putting double-quotes around the value will suffice to treat the `'#'` character as part of the value and not the separator. If the pair has the key word `menu_key` on the left, the value on the right is a key associated with all variable-value pairs on the line. When a menu selection is chosen that contains a key, then all variable-value pairs with that key are posted to the `MOOSDB`. The following configuration will result in the pull-down menu depicted in Figure 82.

```
action  = menu_key=deploy # DEPLOY = true # RETURN = false
action+ = menu_key=deploy # MOOS_MANUAL_OVERIDE=false
action  = RETURN=true
```

The `action+` variant hints to the viewer that a line should be rendered in the pull-down menu separating it from following items.

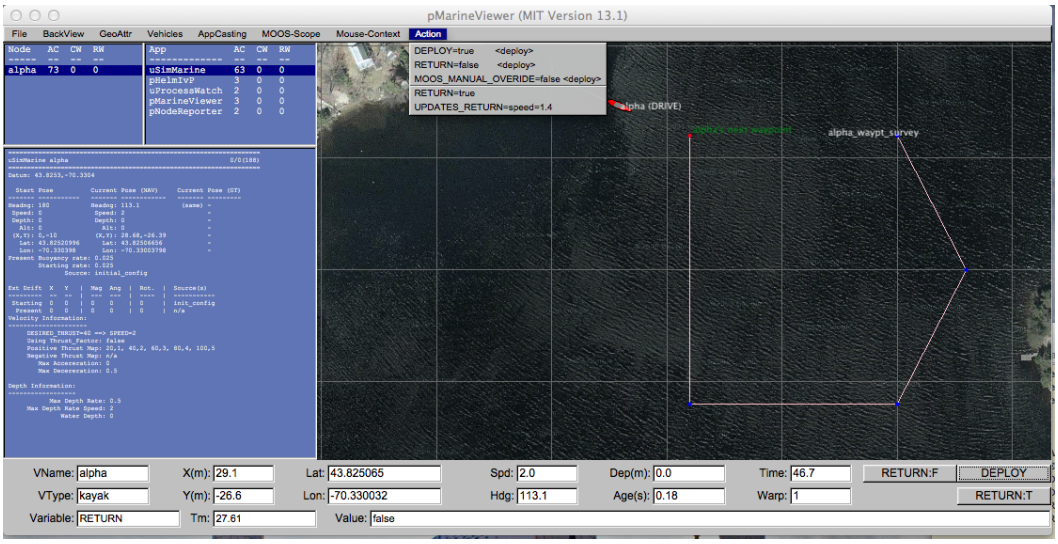


Figure 82: **The Action menu:** The variable value pairs on each menu item may be selected for poking or writing the MOOSDB. The three variable-value pairs above the menu divider will be poked in unison when any of the three are chosen, because they were configured with the same key, `<deploy>`, shown to the right on each item.

The variable-value pair being poked on an action selection will determine the variable type by the following rule of thumb. If the value is non-numerical, e.g., `true`, `one`, it is poked as a string. If

it is numerical it is poked as a double value. If one really wants to poke a string of a numerical nature, the addition of quotes around the value will suffice to ensure it will be poked as a string. For example:

```
action = Vehicle=Nomar # ID="7"
```

As with any other publication to the MOOSDB, if a variable has been previously posted with one type, subsequent posts of a different type will be ignored.

13.8 The Mouse-Context Pull-Down Menu

The Mouse-Context pull-down menu is an optional menu - it will not appear unless it is configured for use. It is used for changing the context of left and right mouse clicks on the operation area.

13.8.1 Generic Poking of the MOOSDB with the Operation Area Position

When the user clicks the left or right mouse in the geo portion of the `pMarineViewer` window, the variables `MVIEWER_LCLICK` and `MVIEWER_RCLICK` are published respectively with the operation area location of the mouse click, and the name of the active vehicle. A left mouse click may result in a publication similar to:

```
MVIEWER_LCLICK = x=19.0,y=57.0,lat=43.8248027,lon=-70.3290334,vname=henry,counter=1
```

A counter is maintained by `pMarineViewer` and is incremented and included on each post. The above style posting presents a generic way to convey to other MOOS applications an operation area position. In this case the other MOOS applications need to conform to this generic output. But, with a bit of further configuration, a similar *custom* post to the MOOSDB is possible to shift the burden of conformity away from the other MOOS applications where typically a user does not have the ability to change the interface.

13.8.2 Custom Poking of the MOOSDB with the Operation Area Position

Custom configuration of mouse clicks is possible by (a) allowing the MOOS variable and value to be defined by the user, and (b) exposing a few macros in the custom specification to embed operation area information. Configuration is done in the MOOS configuration block with entries of the following form:

```
left_context[<key>]  = <var-data-pair>
right_context[<key>] = <var-data-pair>
```

The `left_context` and `right_context` keywords are case insensitive. If no entries are provided, this pull-down menu will not appear. The `<key>` component is optional and allows for groups of variable-data pairs with the same key to be posted together with the same mouse click. This is the selectable *context* in the Mouse-Context pull-down menu. If the `<key>` is empty, the defined posting will be made on all mouse clicks regardless of the grouping, as is the case with `MVIEWER_LCLICK` and `MVIEWER_RCLICK`.

Macros may be embedded in the string to allow the string to contain information on where the user clicked in the operation area. These patterns are: `$(XPOS)` and `$(YPOS)` for the local `x` and `y` position respectively, and `$(LAT)`, and `$(LON)` for the latitude and longitude positions. The pattern `$(IX)` will expand to an index (beginning with zero by default) that is incremented each time a click/poke is made. This index can be configured to start with any desired index with the `lclick_ix_start` and `rclick_ix_start` configuration parameters for the left and right mouse clicks respectively. The following configuration will result in the pull-down menu depicted in Figure 83.

```
left_context[surface_point] = SPOINT = x=$(XPOS), y=$(YPOS), vname=$(VNAME)
left_context[surface_point] = COME_TO_SURFACE = true
left_context[return_point]  = RETURN_POINT = point=$(XPOS),$(YPOS), vname=$(VNAME)
left_context[return_point]  = RETURN_HOME = true
left_context[return_point]  = RETURN_HOME_INDEX = $(IX)
right_context[loiter_point] = LOITER_POINT = lat=$(LAT), lon=$(LON)
right_context[loiter_point] = LOITER_MODE = true
```

Note in the figure that the first menu option is `"no-action"` which shuts off all MOOS pokes associated with any defined groups (keys). In this mode, the `MVIEWER_LCLICK` and `MVIEWER_RCLICK` pokes will still be made, along with any other poke configured without a `<key>`.

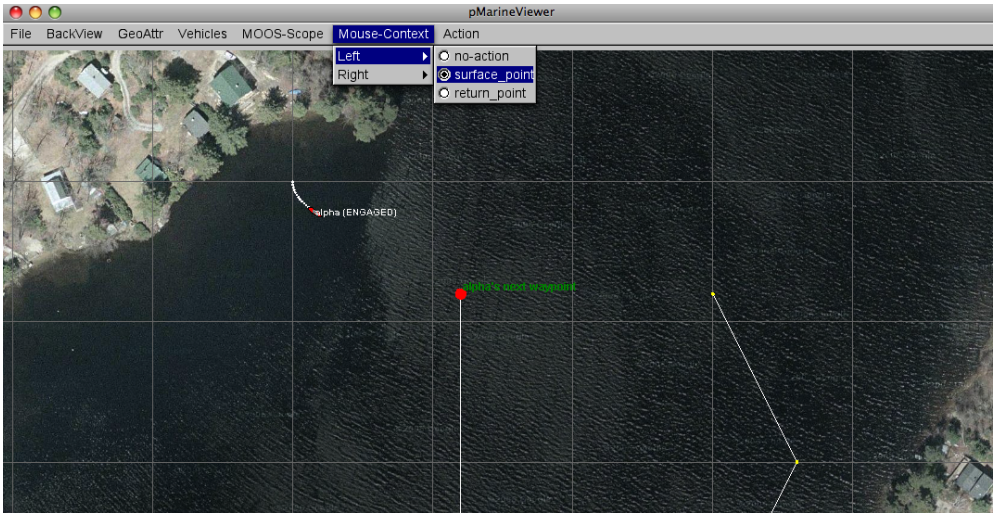


Figure 83: **The Mouse-Context menu:** Keywords selected from this menu will determine which groups of MOOS variables will be poked to the MOOSDB on left or mouse clicks. The variable values may have information embedded indicating the position of the mouse in the operation area at the time of the click.

13.9 The Reference-Point Pull-Down Menu

The "Reference-Point" pull-down menu allows the user to select a reference point other than the datum, the `(0,0)` point in local coordinates. The reference point will affect the data displayed in the `Range` and `Bearing` fields in the viewer window. This feature was originally designed for field experiments when vehicles are being operated from a ship. An operator on the ship running the `pMarineViewer` would receive position reports from the unmanned vehicles as well as the present

position of the ship. In these cases, the ship is the most useful point of reference. Prior versions of this code would allow for a single declaration of the ship name, but the the current version allows for any number of ship names as a possible reference point. This allows the viewer to display the bearing and range between two deployed unmanned vehicles for example. Configuration is done in the MOOS configuration block with entries of the following form:

```
reference_vehicle = vehicle
```

If no such entries are provided, this pull-down menu will not appear. When the menu is present, it looks like that shown in Figure 84. When the reference point is a vehicle with a known heading, the user is able to alter the `Bearing` field from reporting either the relative bearing or absolute bearing. Hot keys are defined for each.

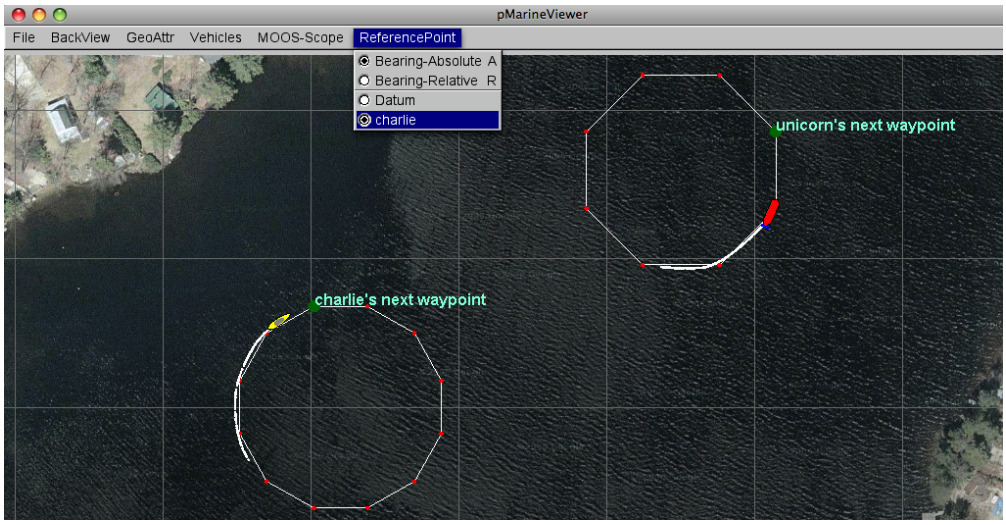


Figure 84: **The Reference-Point menu:** This pull-down menu of the `pMarineViewer` lists the options for selecting a reference point. The reference point determines the values for the `Range` and `Bearing` fields in the viewer for the active vehicle. When the reference point is a vehicle with known heading, the user also may select whether the Bearing is the relative bearing or absolute bearing.

Mini Exercise #2: Poking a Vehicle into the Viewer.

Issues Explored: (1) Posting a MOOS message resulting in a vehicle rendered in pMarineViewer. (2) Erasing and moving the vehicle.

- Try running the Alpha mission again from the Helm documentation. Note that when the simulation is first launched, a kayak-shaped vehicle sits at position `(0,0)`.
- Use the pMarineViewer MOOS-Scope utility to scope on the variable `NODE_REPORT_LOCAL`. Either select `"Add Variable"` from the MOOS-Scope pull-down menu, or type the short-cut key `'a'`. Type the `NODE_REPORT_LOCAL` variable into the pop-up window, and hit Enter. The scope field at the bottom of pMarineViewer should read something like:

  ```
  NODE_REPORT_LOCAL = "NAME=alpha,TYPE=KAYAK,MOOSDB_TIME=2327.07,UTC_TIME=10229133704.48,
  X=0.00,Y=0.00,LAT=43.825300,LON=-70.330400,SPD=0.00,HDG=180.00,YAW=180.00000,DEPTH=0.00,
  LENGTH=4.0,MODE=PARK,ALLSTOP=ManualOverride
  ```

  This is the posting that resulted in the vehicle currently rendered in the pMarineViewer window. This was likely posted by the `pNodeReporter` application, but a node report can be poked directly as well to experiment.
- Using the uPokeDB tool, try poking the MOOSDB as follows:

  ```
  $ uPokeDB alpha.moos NODE_REPORT="NAME=bravo,TYPE=glider,X=100,Y=-90,HDG=88,SPD=1.0,
  UTC_TIME=NOW,DEPTH=92,LENGTH=8"
  ```

  Note the appearance of the glider at position `(100,-90)`.

13.10 Configuration Parameters for pMarineViewer

The blue items in pull-down menus are also available as mission file configuration parameters. The configuration parameter is identical to the pull-down menu text. For example in the BackView menu shown in Figure 70, the menu item `full_screen=true` may also be set in the `pMarineViewer` configuration block verbatim with `full_screen=true`.

13.10.1 Configuration Parameters for the BackView Menu

The parameters in Listing 34 relate to the BackView menu described more fully in Section 13.2. Parameters in blue below correlate to parameters in blue in the pull-down menu. For these parameters, the text in the pull-down menu is identical to a similar entry in the configuration file.

Listing 13.34: Configuration Parameters for pMarineViewer BackView Menu.

`back_shade`:	Shade of gray background when no image is used. Legal value range: [0, 1]. Zero is black, one is white.
`full_screen`:	If true, viewer is in full screen mode (no appcasts, no fields rendered at the bottom). Legal values: true, false. Section 13.2.4.
`hash_delta`:	Sets the hash line spacing. Legal values: 50, 100, 200, 500, 1000. The default is 100. Section 13.2.3.
`hash_shade`:	Shade of hash marks. Legal value range: [0, 1]. Zero is black, one is white. Section 13.2.3.

hash_viewable:	If true, hash lines are rendered over the op area. Legal values: true, false. The default is false. Section 13.2.3.
tiff_file:	Filename of a tiff file background image. Section 13.2.2.
log_the_image:	If true, a request is posted to pLogger to log a copy of the image and info file. Legal values: true, false. The default is false. Section 13.2.2.
tiff_file_b:	Filename of another tiff file background image. Section 13.2.2.
tiff_type:	Use the first tiff image if set to true. Legal values: true, false. The default is true. Section 13.2.2.
tiff_view:	Use the tiff background image if set to true. Otherwise a gray screen is used as a background. Legal values: true, false. The default it true. Section 13.2.2.
view_center:	Sets the center of the viewing image in x,y local coordinates. Legal values: (double,double). The default is (0,0).

13.10.2 Configuration Parameters for the GeoAttributes Menu

The parameters in Listing 35 relate to the GeoAttributes pull-down menu described more fully in Section 13.3. Parameters in blue below correlate to parameters in blue in the pull-down menu. For these parameters, the text in the pull-down menu is identical to a similar entry in the configuration file.

Listing 13.35: Configuration Parameters for pMarineViewer Geometry Menu.

circle_viewable_all:	If false, circles are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.1.
circle_viewable_labels:	If false, circle labels are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.1.
comms_pulse_viewable_all:	If false, comms pulses are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.3.
datum_viewable:	If false, the datum is suppressed from rendering. Legal values: true, false. The default is true. Sections 13.2.2 and 13.3.
datum_color:	The color used for rendering the datum. Legal values: Any color listed in the Colors Appendix. The default is red. Sections 13.2.2 and 13.3.
datum_size:	The size of the point used to render the datum. Legal values: Integers in the range $[1, 10]$. The default is 2. Sections 13.2.2 and 13.3.
drop_point_viewable_all:	If false, drop points are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.5.
drop_point_coords:	Specifies whether the drop point labels are in earth or local coordinates. Legal values are: as-dropped, lat-lon, local-grid. The default is as-dropped. Section 13.3.5.

drop_point_vertex_size: The size of the point used to render a drop point. Legal values: Integers in the range [1, 10]. The default is 2. Section 13.3.5.

grid_viewable_all: If false, grids are suppressed from rendering. Legal values: true, false. The default is true.

grid_viewable_labels: If false, grid labels are suppressed from rendering. Legal values: true, false. The default is true.

grid_viewable_opaqueness: The degree to which grid renderings are opaque. Legal range: [0, 1]. The default is 0.3.

marker A marker may be stated in the configuration file with the same format of the VIEW_MARKER message. Section 13.3.2.

marker_scale: The scale applied to marker renderings. Legal range: [0.1, 100]. The default is 1.0. Section 13.3.2.

marker_viewable_all: If false, markers are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.2.

marker_edge_width: Markers are rendered with an outer black edge. The edge may be set thicker to aid in viewing. Legal values: Integer values in the range [1, 10]. The default is 1. Section 13.3.2.

marker_viewable_labels: If false, marker labels are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.2.

oparea_viewable_all: If false, oparea lines are suppressed from rendering. Legal values: true, false. The default is true.

oparea_viewable_labels: If false, oparea label is suppressed from rendering. Legal values: true, false. The default is true.

point_viewable_all: If false, points are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.1.

point_viewable_labels: If false, point labels are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.1.

polygon_viewable_all: If false, polygons are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.1.

polygon_viewable_labels: If false, polygon labels are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.1.

range_pulse_viewable_all: If false, range pulses are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.3.

seglist_viewable_all: If false, seglists are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.1.

seglist_viewable_labels: If false, seglist labels are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.1.

vector_viewable_all: If false, vectors are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.1.

vector_viewable_labels: If false, vector labels are suppressed from rendering. Legal values: true, false. The default is true. Section 13.3.1.

13.10.3 Configuration Parameters for the Vehicles Menu

The parameters in Listing 36 relate to the Vehicles pull-down menu described more fully in Section 13.4. Parameters in blue below correlate to parameters in blue in the pull-down menu. For these parameters, text in the pull-down menu is identical to a similar entry in the configuration file.

Listing 13.36: Configuration Parameters for pMarineViewer Vehicles Pull-Down Menu.

bearing_lines_viewable:	If false, bearing lines will be suppressed from rendering. Legal values: true, false. The default is true.
center_view:	Sets the pan position to be either directly above the active vehicle, or the average of all vehicles. Legal values: active, average. The default is neither, resulting in the pan position being set to either (0,0) or set via other configuration parameters. Section 13.4.5.
stale_nodraw_thresh:	Number of seconds after which a vehicle report will be considered stale enough to no longer draw. If stale_vehicles_viewable is true however, the vehicle will be drawn anyway regardless of this setting. Legal values: Any non-negative number. The default is 300. Section 13.4.2.
stale_report_thresh:	Number of seconds after which a vehicle report will be considered stale. Legal values: Any non-negative number. The default is 60. Section 13.4.2.
stale_vehicles_viewable:	If false, stale vehicles are suppressed from rendering. Staleness occurs after 300 seconds by default. Legal values: true, false. The default is true. Section 13.4.2.
trails_color:	The color of trail points rendered behind vehicles to indicate recent vehicle position history. Legal values: Any color listed in the Colors Appendix. The default is white. Section 13.4.6.
trails_connect_viewable:	If true the vehicle trail points are each connected by a line. Useful when node reports have large gaps in time. Legal values: true, false. The default is true. Section 13.4.6.
trails_length:	The number of points retained for the rendering of vehicle trails. Legal values: Integers in the range $[1, 100000]$. The default is 100. Section 13.4.6.
trails_point_size:	The size of the points rendering the vehicle trails. Legal values: Integers in the range $[1, 10]$. The default is 1. Section 13.4.6.
trails_viewable:	If false, vehicle trails are suppressed from rendering. Legal values: true, false. The default is true. Section 13.4.6.
vehicles_active_color:	The color of the active vehicle (the one who's data is being shown in the bottom data fields). Legal values: Any color listed in the Colors Appendix. The default is red. Section 13.4.4.
vehicles_inactive_color:	The color of inactive vehicles. Legal values: Any color listed in the Colors Appendix. The default is yellow. Section 13.4.4.
vehicles_shape_scale:	The scale factor applied to vehicle size rendering. Legal values in the range: $[0.1, 100]$. The default is 1.0. Section 13.4.3.

vehicles_name_mode:	Sets the mode for rendering the vehicle label. Legal values are: names, names+mode, names+shortmode, names+depth, off. The default is names+shortmode. Section 13.4.1.
vehicles_name_color:	Sets the color for rendering the vehicle label. Legal values are any color in Appendix C. The default is white. Section 13.4.4.
vehicles_viewable:	If false, vehicles are suppressed from rendering. Legal values: true, false. The default is true. Section 13.4.

13.10.4 Configuration Parameters for the AppCast Menu

The parameters in Listing 37 relate to the AppCast pull-down menu described more fully in Section 13.5. Parameters in blue below correlate to parameters in blue in the pull-down menu. For these parameters, text in the pull-down menu is identical to a similar entry in the configuration file.

Listing 13.37: Configuration Parameters for pMarineViewer AppCast Pull-Down Menu.

appcast_color_scheme:	The color scheme used in all three appcasting panes, affecting background color and font color. Legal values: white, indigo, beige. The default is indigo. Section 13.5.5.
appcast_font_size:	The font size uses in the *appcast* pane of the set of appcasting panes. Legal values: large, medium, small, xsmall. The default is small. Section 13.5.4.
appcast_height:	The height of the appcasting bottom pane as a percentage of the total `pMarineViewer` window height. Legal values: [30, 35, 40, 45,..., 85, 90]. The default is 75. Section 13.5.2.
appcast_viewable:	If true, the appcasting set of panes are rendered on the left side of the viewer. Legal values: true, false. The default is true. Section 13.5.1.
appcast_width:	The width of the appcasting panes as a percentage of the total `pMarineViewer` window width. Legal values: [20, 25, 30, 35,..., 65, 70]. The default is 30. Section 13.5.2.
nodes_font_size:	the font size uses in the *nodes* pane of the set of appcasting panes. Legal values: large, medium, small, xsmall. The default is medium. Section 13.5.4.
procs_font_size:	The font size uses in the *procs* pane of the set of appcasting panes. Legal values: large, medium, small, xsmall. The default is medium. Section 13.5.4.
refresh_mode:	Determines the manner in which appcast requests are sent to apps. Legal values: paused, events, streaming. The default is events. Section 13.5.3.

13.10.5 Configuration Parameters for the Scope, MouseContext and Action Menus

Listing 13.38: Configuration Parameters the Scope, MouseContext and Action Menus.

scope:	A comma separated list of MOOS variables to scope. Section 13.6.
oparea:	A specification of the operation area boundary for optionally rendering.
button_one:	A configurable command and control button. Section 13.1.5.
button_two:	A configurable command and control button. Section 13.1.5.
button_three:	A configurable command and control button. Section 13.1.5.
button_four:	A configurable command and control button. Section 13.1.5.
action:	A MOOS variable-value pair for posting, available under the Action pull-down menu. Section 13.7.
left_context:	Allows the custom configuration of left mouse click context. Section 13.8.
right_context:	Allows the custom configuration of right mouse click context. Section 13.8.
lclick_ix_start:	Starting index for the left mouse index macro. Section 13.8.
rclick_ix_start:	Starting index for the right mouse index macro. Section 13.8.

13.11 Publications and Subscriptions for pMarineViewer

The interface for pMarineViewer, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pMarineViewer --interface or -i
```

13.11.1 Variables Published by pMarineViewer

It is possible to configure pMarineViewer to poke the MOOSDB via either the Action pull-down menu (Section 13.7), or via configurable GUI buttons (Section 13.1.5). It may also publish to the MOOSDB variables configured to mouse clicks (Section 13.8). So the list of variables that pMarineViewer publishes is somewhat user dependent, but the following few variables may be published in all configurations.

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section **??**.
- APPCAST_REQ_<COMMUNITY>: As an appcast viewer, pMarineViewer also generates outgoing appcast requests to MOOS communities it is aware of, including its own MOOS community. These postings are typically bridged to the other named MOOS community with the variable renamed simply to APPCAST_REQ when it arrives in the other community.
- HELM_MAP_CLEAR: This variable is published once when the viewer connects to the MOOSDB. It is used in the pHelmIvP application to clear a local buffer used to prevent successive identical publications to its variables.
- MVIEWER_LCLICK: When the user clicks the left mouse button, the position in local coordinates, along with the name of the active vehicle is reported. This can be used as a command and control hook as described in Section 13.8.
- MVIEWER_RCLICK: This variable is published when the user clicks with the right mouse button. The same information is published as with the left click.
- PLOGGER_CMD: This variable is published with a "COPY_FILE_REQUEST" to log a copy of the image and info file, only if log_the_image is set to true. Section 13.2.2.

13.11.2 Variables Subscribed for by pMarineViewer

- APPCAST: As an appcast *viewer*, pMarineViewer also subscribes for appcasts from other other applications and communities to provide the content for its own viewing capability.
- APPCAST_REQ: As an appcast enabled MOOS application, pMarineViewer also subscribes for appcast requests. Each incoming message is a request to generate and post a new appcast report, with reporting criteria, and expiration. Section **??**.
- NODE_REPORT: This is the primary variable consumed by pMarineViewer for collecting vehicle position information.
- NODE_REPORT_LOCAL: This serves the same purpose as the above variable. In some simulation cases this variable is used.
- TRAIL_RESET: When the viewer receives this variable it will clear the history of trail points associated with each vehicle. This is used when the viewer is run with a simulator and the vehicle position is reset and the trails become discontinuous.
- VIEW_CIRCLE: A string representation of an XYCircle object.
- VIEW_COMMS_PULSE: A string representation of an XYCommsPulse object.
- VIEW_GRID: A string representation of a XYConvexGrids object.
- VIEW_GRID_CONFIG: A string representation of a XYGrid configuration.
- VIEW_GRID_DELTA: A string representation of a XYGrid configuration.
- VIEW_POINT: A string representation of an XYPoint object.
- VIEW_POLYGON: A string representation of an XYPolygon object.
- VIEW_SEGLIST: A string representation of an XYSegList object.
- VIEW_MARKER: A string designation of a marker type, size and location.
- VIEW_RANGE_PULSE: A string representation of an XYRangePulse object.
- VIEW_VECTOR: A string representation of an XYVector object

14 uTimerScript: Scripting Events to the MOOSDB

14.1 Overview

The `uTimerScript` application allows the user to script a set of pre-configured posts to a MOOSDB. In its most basic form, it may be used to initialize a set of variables to the MOOSDB, and immediately terminate itself if a quit event is included. The following configuration block, if placed in the alpha example mission, would mimic the posts to the MOOSDB behind the `DEPLOY` button, simply disabling manual control, deploying the vehicle and quitting the script: Listing 14.39.

Listing 39: A Simple Timer Script.

```
ProcessConfig = uTimerScript
{
  event = var=MOOS_MANUAL_OVERIDE, val=false
  event = var=DEPLOY, val=true
  event = quit
}
```

Additionally, `uTimerScript` may be used with the following advanced functions:

- Each entry in the script may be scheduled to occur after a specified amount of elapsed time.
- Event timestamps may be given as an exact point in time relative to the start of the script, or a range in times with the exact time determined randomly at run-time.
- The execution of the script may be paused, or fast-forwarded a given amount of time, or forwarded to the next event on the script by writing to a MOOS variable.
- The script may be conditionally paused based on user defined logic conditions over one or more MOOS variables.
- The variable value of an event may also contain information generated randomly.
- The script may be reset or repeated any given number of times.
- The script may use its own time warp, which can be made to vary randomly between script executions.

In short, `uTimerScript` may be used to effectively simulate the output of other MOOS applications when those applications are not available. A few examples are provided, including a simulated GPS unit and a crude simulation of wind gusts.

14.2 Using uTimerScript

Configuring a script minimally involves the specification of one or more events, with an event comprising of a MOOS variable and value to be posted and an optional time at which it is to be posted. Scripts may also be reset on a set policy, or from a trigger by an external process.

14.2.1 Configuring the Event List

The event list or script is configured by declaring a set of event entries with the following format:

```
event = var=<MOOSVar>, val=<value>, [time=<time-of-event>]
```

The keywords event, var, val, and time are not case sensitive, but the values <moos-variable> and <var-value> are case sensitive. The <var-value> type is posted either as a string or double based on the following heuristic: if the <var-value> has a numerical value it is posted as a double, and otherwise posted as a string. If one wants to post a string with a numerical value, putting quotes around the number suffices to have it posted as a string. Thus val=99 posts a double, but var="99" posts a string. If a string is to be posted that contains a comma such as "apples, pears", one must put the quotes around the string to ensure the comma is interpreted as part of <var-value>. The value field may also contain one or more macros expanded at the time of posting, as described in Section 14.4.

14.2.2 Setting the Event Time or Range of Event Times

The value of <time-of-event> is given in seconds and must be a numerical value greater or equal to zero. The time represents the amount of elapsed time since the uTimerScript was first launched and un-paused. The list of events provided in the configuration block need not be in order - they will be ordered by the uTimerScript utility. The <time-of-event> may also be specified by a interval of time, e.g., time=0:100, such that the event may occur at some point in the range with uniform probability. The only restrictions are that the lower end of the interval is greater or equal to zero, and less than or equal to the higher end of the interval. By default the timestamps are calculated once from their specified interval, at the the outset of uTimerScript. The script may alternatively be configured to recalculate the timestamps from their interval each time the script is reset, by setting the shuffle parameter to true. This parameter, and resetting in general, are described in the next Section 14.2.3.

14.2.3 Resetting the Script

The timer script may be reset to its initial state, resetting the stored elapsed-time to zero and marking all events in the script as pending. This may occur either by cueing from an event outside uTimerScript, or automatically from within uTimerScript. Outside-cued resets can be triggered by posting UTS_RESET with the value ="reset", or "true". The reset_var parameter names a MOOS variable that may be used as an alternative to UTS_RESET. It has the format:

```
reset_var = <moos-variable> // Default is UTS_RESET
```

The script may be also be configured to auto-reset after a certain amount of time, or immediately after all events are posted, using the reset_time parameter. It has the format:

```
reset_time = <time-or-condition> // Default is "none"
```

The <time-or-condition> may be set to "all-posted" which will reset after the last event is posted. If set to a numerical value greater than zero, it will reset after that amount of elapsed time, regardless of whether or not there are pending un-posted events. If set to "none", the default, then no automatic resetting is performed. Regardless of the reset_time setting, prompted resets via the UTS_RESET variable may take place when cued.

The script may be configured to accept a hard limit on the number of times it may be reset. This is configured using the reset_max parameter and has the following format:

```
reset_max = <amount> // Default is "nolimit"
```

The <amount> specified may be any number greater or equal to zero, where the latter, in effect, indicates that no resets are permitted. If unlimited resets are desired (the default), the case insensitive argument "unlimited" or "any" may be used.

The script may be configured to recalculate all event timestamps specified with a range of values whenever the script is reset. This is done with the following parameter:

```
shuffle = true // Default is "false"
```

The script may be configured to reset or restart each time it transitions from a situation where its conditions are not met to a situation where its conditions are met, or in other words, when the script is "awoken". The use of logic conditions is described in more detail in Section 14.3.1. This is done with the following parameter:

```
upon_awake = restart // Default is "n/a", no action
```

Note that this does not apply when the script transitions from being paused to un-paused as described in Section 14.3.1. See the example in Section 14.9.1 for a case where the upon_awake feature is handy.

14.3 Script Flow Control

The script flow may be affected in a number of ways in addition to the simple passage of time. It may be (a) paused by explicitly pausing it, (b) implicitly paused by conditioning the flow on one or more logic conditions, (c) fast-forwarded directly to the next scheduled event, or fast-forwarded some number of seconds. Each method is described in this section.

14.3.1 Pausing the Timer Script

The script can be paused at any time and set to be paused initially at start time. The paused parameter affects whether the timer script is actively unfolding at the outset of launching uTimerScript. It has the following format:

```
paused = <Boolean>
```

The keyword paused and the string representing the Boolean are not case sensitive. The Boolean simply must be either "true" or "false". By setting paused to true, the elapsed time calculated by uTimerScript is paused and no variable-value pairs will be posted. When un-paused the elapsed time begins to accumulate and the script begins or resumes unfolding. The default value of paused is false.

The script may also be paused through the MOOS variable UTS_PAUSE which may be posted by some other MOOS application. The values recognized are "true", "false", or "toggle", all case insensitive. The name of this variable may be substituted for a different one with the pause_var parameter in the uTimerScript configuration block. It has the format:

```
pause_var = <MOOSVar> // Default is UTS_PAUSE
```

If multiple scripts are being used (with multiple instances of uTimerScript connected to the MOOSDB), setting the pause_var to a unique variable may be needed to avoid unintentionally pausing or un-pausing multiple scripts with single write to UTS_PAUSE.

14.3.2 Conditional Pausing of the Timer Script and Atomic Scripts

The script may also be configured to condition the "paused-state" to depend on one or more logic conditions. If conditions are specified in the configuration block, the script must be both un-paused as described above in Section 14.3.1, and all specified logic conditions must be met in order for the script to begin or resume proceeding. The logic conditions are configured as follows:

```
condition = <logic-expression>
```

The `<logic-expression>` syntax is described in Appendix A, and may involve the simple comparison of MOOS variables to specified literal values, or the comparison of MOOS variables to one another. See the script configuration in Section 14.9.1 for one example usage of logic expressions.

An *atomic* script is one that does not check conditions once it has posted its first event, and prior to posting its last event. Once a script has started, it is treated as unpausable with respect to the the logic conditions. This is configured with:

```
script_atomic = <Boolean>
```

It can however be paused and unpaused via the pause variable, e.g., `UTS_PAUSE`, as described in Section 14.3.1. If the logic conditions suddenly fail in an atomic script midway, the check is simply postponed until after the script completes and is perhaps reset. If the conditions in the meanwhile revert to being satisfied, then no interruption should be observable.

14.3.3 Fast-Forwarding the Timer Script

The timer script, when un-paused, moves forward in time with events executed as their event times arrive. However, the script may be moved forwarded by writing to the MOOS variable `UTS_FORWARD`. If the value received is zero (or negative), the script will be forwarded directly to the point in time at which the next scheduled event occurs. If the value received is positive, the elapsed time is forwarded by the given amount. Alternatives to the MOOS variable `UTS_FORWARD` may be configured with the parameter:

```
forward_var = <MOOSVar> // Default is UTS_FORWARD
```

If multiple scripts are being used (with multiple instances of `uTimerScript` connected to the MOOSDB), setting the `forward_var` to a unique variable may be needed to avoid unintentionally fast forwarding multiple scripts with single write to `UTS_FORWARD`.

14.3.4 Quitting the Timer Script

The timer script may be configured with a special event, the *quit* event, resulting in disconnection with the MOOSDB and a process exit. This is done with the configuration:

```
event = quit [time=<time-of-event>]
```

Before quitting, a final posting to the MOOSDB is made with the variable `EXITED_NORMALLY`. The value is `"uTimerScript"`, or its alias if an alias was used. This indicates to any other watchdog process, such as `uProcessWatch`, that the exiting of this script is not a reason for concern. When `uTimerScript` receives its own posting in the next incoming mail, it assumes all pending posts have been made and will then quit.

14.4 Macro Usage in Event Postings

Macros may be used to add a dynamic component to the value field of an event posting. This substantially expands the expressive power and possible uses of the `uTimerScript` utility. Recall that the components of an event are defined by:

```
event = var=<MOOSVar>, val=<var-value>, time=<time-of-event>
```

The `<var-value>` component may contain a macro of the form `$[MACRO]`, where the macro is either one of a few built-in macros available, or a user-defined macro with the ability to represent random variables. Macros may also be combined in simple arithmetic expressions to provide further expressive power. In each case, the macro is expanded at the time of the event posting, typically with different values on each successive posting.

14.4.1 Built-In Macros Available

There are five built-in macros available: `$[DBTIME]`, `$[UTCTIME]`, `$[COUNT]`, `$[TCOUNT]`, and `$[IDX]`. The first macro expands to the estimated time since the MOOSDB started, similar to the value in the MOOS variable `DB_UPTIME` published by the MOOSDB. An example usage:

```
event = var=DEPLOY_RECEIVED, val=$[DBTIME], time=10:20
```

The `$[UTCTIME]` macro expands to the UTC time at the time of the posting. The `$[COUNT]` macro expands to the integer total of all posts thus far in the current execution of the script, and is reset to zero when the script resets. The `$[TCOUNT]` macro expands to the integer total of all posts thus far since the application began, i.e., it is a running total that is not reset when the script is reset.

The `$[DBTIME]`, `$[UTCTIME]`, `$[COUNT]`, and `$[TCOUNT]` macros all expand to numerical values, which if embedded in a string, will simply become part of the string. If the value of the MOOS variable posting is solely this macro, the variable type of the posting is instead a double, not a string. For example `val=$[DBTIME]` will post a type double, whereas `val="time:$[DBTIME]"` will post a type string.

The `$[IDX]` macro is similar to the `$[COUNT]` macro in that it expands to the integer value representing an event's count or index into the sequence of events. However, it will always post as a string and will be padded with zeros to the left, e.g., `"000"`, `"001"`, ... and so on.

14.4.2 User Configured Macros with Random Variables

Further macros are available for use in the `<var-value>` component of an event, defined and configured by the user, and based on the idea of a random variable. In short, the macro may expand to a numerical value chosen within a user specified range, and recalculated according to a user-specified policy. The general format is:

```
rand_var = varname=<variable>, min=<value>, max=<value>, key=<key_name>
```

The `<variable>` component defines the macro name. The `<low_value>` and `<high_value>` components define the range from which the random value will be chosen uniformly. The `<key_name>` determines when the random value is reset. The following three key names are significant: `"at_start"`,

"at_reset", and "at_post". Random variables with the key name "at_start" are assigned a random value only at the start of the uTimerScript application. Those with the "at_reset" key name also have their values re-assigned whenever the script is reset. Those with the "at_post" key name also have their values re-assigned after any event is posted.

14.4.3 Support for Simple Arithmetic Expressions with Macros

Macros that expand to numerical values may be combined in simple arithmetic expressions with other macros or scalar values. The general form is:

```
{<value> <operator> <value>}
```

The <value> components may be either a scalar or a macro, and the <operator> component may be one of '+', '-', '*', '/'. Nesting is also supported. Below are some examples:

```
{$[FOOBAR] * 0.5}
{-2-$[FOOBAR]}
{$[APPLES] + $[PEARS]}
{35 / {$[FOOBAR]-2}}
{$[DBTIME] - {35 / {$[UTCTIME]+2}}}
```

If a macro should happen to expand to a string rather than a double (numerical) value, the string evaluates to zero for the sake of the remaining evaluations.

14.5 Time Warps, Random Time Warps, and Restart Delays

A time warp and initial start delay may be optionally configured into the script to change the event schedule without having to edit all the time entries for each event. They may also be configured to take on a new random value at the outset of each script execution to allow for simulation of events in nature or devices having a random component.

14.5.1 Random Time Warping

The time warp is a numerical value in the range $(0, \infty]$, with a default value of 1.0. Lower values indicate that time is moving more slowly. As the script unfolds, a counter indicating "elapsed_time" increases in value as long as the script is not paused. The "elapsed_time" is multiplied by the time warp value. The time warp may be specified as a single value or a range of values as below:

```
time_warp = <value>
time_warp = <low-value>:<high-value>
```

When a range of values is specified, the time warp value is calculated at the outset, and re-calculated whenever the script is reset. See the example in Section 14.9.2 for a use of random time warping to simulate random wind gusts.

14.5.2 Random Initial Start and Reset Delays

A start delay may be provided with the delay_start parameter, given in seconds in the range $[0, \infty]$, with a default value of 0. The effect of having a non-zero delay of n seconds is to have elapsed_time=n at the outset of the script, on the first time through the script only. Thus a delay of n seconds combined with a time warp of 0.5 would result in observed delay of $2 * n$ seconds. The start delay may be specified as a single value or a range of values as below:

```
delay_start = <value>
delay_start = <low-value>:<high-value>
```

To specify a delay applied at the beginning if the script *after a reset*, use the delay_reset parameter instead.

```
delay_reset = <value>
delay_reset = <low-value>:<high-value>
```

When a range of values is specified, the start ore reset delay value is calculated at the outset, and re-calculated whenever the script is reset. See the example in Section 14.9.1 for a use of random start delays to the simulate the delay in acquiring satellite fixes in a GPS unit on an UUV coming to the surface.

14.5.3 Status Messages Posted to the MOOSDB by uTimerScript

The uTimerScript periodically publishes a string to the MOOS variable UTS_STATUS indicating the status of the script. This variable will be published on each iteration if one of the following conditions is met: (a) two seconds has passed since the previous status message posted, or (b) an event has been been posted, or (c) the paused state has changed, or (d) the script has been reset, or (e) the state of script logic conditions has changed. A posting may look something like:

```
UTS_STATUS = "name=RND_TEST, elapsed_time=2.00, posted=1, pending=5, paused=false,
              conditions_ok=true, time_warp=3, start_delay=0, shuffle=false,
              upon_awake=restart, resets=2/5"
```

In this case, the script has posted one of six events (posted=1, pending=5). It is actively unfolding, since paused=false (Section 14.3.1) and conditions_ok=true (Section 14.3.2). It has been reset twice out of a maximum of five allowed resets (resets=2/5, Section 14.2.3). Time warping is being deployed (time_warp=3, Section 14.5), there is no start delay in use (start_delay=0, Section 14.5.2). The shuffle feature is turned off (shuffle=false, Section 14.2.3). The script is not configured to reset upon re-entering the un-paused state (awake_reset=false, Section 14.2.3).

When multiple scripts are running in the same MOOS community, one may want to take measures to discern between the status messages generated across scripts. One way to do this is to use a unique MOOS variable other than UTS_STATUS for each script. The variable used for publishing the status may be configured using the status_var parameter. It has the following format:

```
status_var = <MOOSVar> // Default is UTS_STATUS
```

Alternatively, a unique name may be given to each to each script. All status messages from all scripts would still be contained in postings to UTS_STATUS, but the different script output could be discerned by the name field of the status string. The script name is set with the following format.

```
script_name = <string> // Default is "unnamed"
```

14.6 Terminal and AppCast Output

The script configuration and progress of script execution may also be monitored from an open console window where `uTimerScript` is launched, or through an appcast viewer. Example output is shown below in Listing 40. On line 2, the name of the local community or vehicle name is listed on the left. On the right, `"0/0(450)` indicates there are no configuration or run warnings, and the current iteration of `uFldTimerScript` is 450.

Lines 4-16: Script Configuration

Lines 4-11 show the script configuration. Line 5 shows the number of elements in the script and in parentheses the last element to have been posted. Line 6 shows the number of times the script has restarted. Line 7 shows the present time warp and the range of time warps possible on each script restart in brackets (Section 14.5.1. Lines 8-9 show the delay applied at the start and after a script reset (Section 14.5.2). Line 10 indicates the script is presently not paused (Section 14.3.1). Line 11 indicates the script presently meets any prevailing logic conditions (Section 14.3.2). Lines 13-16 show that there are two random variables defined for this script that may be used in event definitions. They are both uniform random variables. The first varies over possible directions, and the second over possible speed magnitudes. Section 14.4.2.

Listing 40 - Example uTimerScript console and appcast output.

```
 1  ============================================================================
 2  uTimerScript charlie                                              0/0(450)
 3  ============================================================================
 4  Current Script Information:
 5      Elements: 10(8)
 6       Reinits: 2
 7     Time Warp: 1.09 [0.2,2]
 8   Delay Start: 0
 9   Delay Reset: 23.66 [10,60]
10        Paused: false
11  ConditionsOK: true
12
13  RandomVar  Type     Min  Max  Parameters
14  ---------  -------  ---  ---  ----------
15  ANG        uniform  0    359
16  MAG        uniform  1.5  3.5
17
18  P/Tot  P/Loc  T/Total   T/Local  Variable/Var
19  -----  -----  --------  -------  ------------
20  19     9      196.77    72.15    DRIFT_VECTOR_ADD = 193,-0.6
21  20     0      219.42    24.08    DRIFT_VECTOR_ADD = 12,0.4
22  21     1      220.93    25.71    DRIFT_VECTOR_ADD = 12,0.4
23  22     2      222.95    27.91    DRIFT_VECTOR_ADD = 12,0.4
```

```
24  23     3      224.96    30.09    DRIFT_VECTOR_ADD = 12,0.4
25  24     4      226.46    31.73    DRIFT_VECTOR_ADD = 12,0.4
26  25     5      228.47    33.91    DRIFT_VECTOR_ADD = 12,-0.4
27  26     6      230.49    36.10    DRIFT_VECTOR_ADD = 12,-0.4
28
29  ======================================================================
30  Most Recent Events (3):
31  ======================================================================
32  [192.78]: Script Re-Start. Warp=0.48922, DelayStart=0.0, DelayReset=54.1
33  [44.80]: Script Re-Start. Warp=1.61692, DelayStart=0.0, DelayReset=18.9
34  [0.51]: Script Start. Warp=1, DelayStart=0.0, DelayReset=0.0
```

Lines 18-27: Recent Script Postings

Lines 18-27 show recent postings to the MOOSDB by the script. The first column shows the total postings so far for the script. The second column shows the index within the script. In the above example, there are ten elements in the script. The most recent posting on line 27, shows the script has been reset twice and the most recent posting is of the seventh element of the script (index 6). The third column shows the total time since script started, and the fourth column shows the time since the script was re-started. Note the time delay between lines 20 and 21, due to the `delay_reset` shown on line 9. The last column shows the actual variable value pair posted.

Lines 29-34: Recent Events

Lines 29-34 show recent events (other than event postings). In this case it shows the script has been started, and re-started twice. Notice the delay reset on line 32 is different than that on line 9. The delay reset time of 23.66 seconds shown on line 9 is the delay reset to be applied on the *next* reset.

14.7 Configuration File Parameters for uTimerScript

The following parameters are defined for `uTimerScript`. A more detailed description is provided in other parts of this section. Parameters having default values are indicated.

Listing 41: Configuration Parameters for uTimerScript.

`condition`: A logic condition that must be met for the script to be un-paused. Section 14.3.2.

`delay_reset`: Number of seconds added to each event time, on each script reset. Legal values: any non-negative numerical value, or range of values separated by a colon. The default is zero. Section 14.5.2.

`delay_start`: Number of seconds, or range of seconds, added to each event time, on first pass only. Legal values: any non-negative numerical value, or range of values separated by a colon. The default is zero. Section 14.5.2.

`event`: A description of a single event in the timer script. Section 14.2.1.

`forward_var`: A MOOS variable for taking cues to forward time. The default is `UTS_FORWARD`). Section 14.3.3.

paused:	A Boolean indicating whether the script is paused upon launch. Legal values: true, false. The default is false. Section 14.3.1.
pause_var:	A MOOS variable for receiving pause state cues (UTS_PAUSE). Section 14.3.1.
rand_var:	A declaration of a random variable macro to be expanded in event values. Section 14.4.2.
reset_max:	The maximum amount of resets allowed. Legal values: any non-negative integer, or the string "nolimit". The default is "nolimit". Section 14.2.3.
reset_time:	The time or condition when the script is reset Legal values: Any non-negative number, or the strings "none", "all-posted", or "end". The default is "none". Section 14.2.3.
reset_var:	A MOOS variable for receiving reset cues. The default is UTS_RESET.
script_atomic:	When true, a started script will complete if conditions suddenly fail. Legal values: true, false. The default is false.
script_name:	Unique (hopefully) name given to this script. The default is "unnamed".
shuffle:	If true, timestamps are recalculated on each reset of the script. Legal values: true, false. The default is true. Section 14.2.3.
status_var:	A MOOS variable for posting status summary. The default is UTS_STATUS. Section 14.5.3
time_warp:	Rate at which time is accelerated in executing the script. Legal values: any non-negative number. The default is zero. Section 14.5.
upon_awake:	Reset or re-start the script upon conditions being met after failure ("n/a"). Section 14.2.3.
verbose:	If true, progress output is generated to the console (true).

14.8 Publications and Subscriptions for uTimerScript

The interface for uTimerScript, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uTimerScript --interface
```

14.8.1 Variables Published by uTimerScript

The primary output of uTimerScript to the MOOSDB is the set of configured events, but one other variable is published on each iteration, and another upon purposeful exit with the event=quit event configuration.

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 14.6.
- EXITED_NORMALLY: A posting made when the script contains and executes a event=quit event, to let other applications know that the disconnection of uTimerScript is not a concern for alarm.
- UTS_STATUS: A status string of script progress. Section 14.5.3.

14.8.2 Variables Subscribed for by uTimerScript

The `uTimerScript` application will subscribe for the following four MOOS variables to provide optional control over the flow of the script by the user or other MOOS processes:

- `APPCAST_REQ`: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section **??**.
- `EXITED_NORMALLY`: When `uTimerScript` receives its own posting, it is assumed that all outgoing posts needed to be made before quitting have been received by the MOOSDB. Upon this receipt åuTimerScript will quit. See Section 14.3.4.
- `UTS_NEXT`: When received with the value `"next"`, the script will fast-forward in time to the next event. See Section 14.3.3.
- `UTS_RESET`: When received with the value of either `"true"` or `"reset"`, the timer script will be reset. See Section 14.2.3.
- `UTS_FORWARD`: When received with a numerical value greater than zero, the script will fast-forward by the indicated time. See Section 14.3.3.
- `UTS_PAUSE`: When received with the value of `"true"`, `"false"`, `"toggle"`, the script will change its pause state correspondingly. See Section 14.3.1.

In addition to the above MOOS variables, `uTimerScript` will subscribe for any variables involved in logic conditions, described in Section 14.3.2.

14.8.3 An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command-line:

```
$ uTimerScript --example
```

This will show the output shown in Listing 42 below.

42 - Example configuration of the uTimerScript application.

```
 1  ============================================================
 2  uTimerScript Example MOOS Configuration
 3  ============================================================
 4  Blue lines:     Default configuration
 5
 6  ProcessConfig = uTimerScript
 7  {
 8    AppTick   = 4
 9    CommsTick = 4
10
11    // Logic condition that must be met for script to be unpaused
12    condition      = WIND_GUSTS = true
13    // Seconds added to each event time, on each script pass
14    delay_reset    = 0
15    // Seconds added to each event time, on first pass only
16    delay_start    = 0
17    // Event(s) are the key components of the script
18    event          = var=SBR_RANGE_REQUEST, val="name=archie", time=25:35
```

```
19    // A MOOS variable for taking cues to forward time
20    forward_var    = UTS_FORWARD   // or other MOOS variable
21    // If true script is paused upon launch
22    paused         = false    // or {true}
23    // A MOOS variable for receiving pause state cues
24    pause_var      = UTS_PAUSE     // or other MOOS variable
25    // Declaration of random var macro expanded in event values
26    randvar        = varname=ANG, min=0, max=359, key=at_reset
27    // Maximum number of resets allowed
28    reset_max      = nolimit  // or in range [0,inf)
29    // A point when the script is reset
30    reset_time     = none     // or {all-posted} or range (0,inf)
31    // A MOOS variable for receiving reset cues
32    reset_var      = UTS_RESET     // or other MOOS variable
33    // If true script will complete if conditions suddenly fail
34    script_atomic  = false    // or {true}
35    // A hopefully unique name given to the script
36    script_name    = unnamed
37    // If true timestamps are recalculated on each script reset
38    shuffle        = true
39    // If true progress is generated to the console
40    verbose        = true     // or {false}
41    // Reset or restart script upon conditions being met after failure
42    upon_awake     = n/a      // or {reset,resstart}
43    // A MOOS variable for posting the status summary
44    status_var     = UTS_STATUS    // or other MOOS variable
45    // Rate at which time is accelerated in executing the script
46    time_warp      = 1
47  }
```

14.9 Examples

The examples in this section demonstrate the constructs thus far described for the `uTimerScript` application. In each case, the use of the script obviated the need for developing and maintaining a separate dedicated MOOS application.

14.9.1 A Script Used as Proxy for an On-Board GPS Unit

Typical operation of an underwater vehicle includes the periodic surfacing to obtain a GPS fix to correct navigation error accumulated while under water. A GPS unit that has been out of satellite communication for some period normally takes some time to re-acquire enough satellites to resume providing position information. From the perspective of the helm and configuring an autonomy mission, it is typical to remain at the surface only long enough to obtain the GPS fix, and then resume other aspects of the mission at-depth.

Consider a situation as shown in Figure 85, where the autonomy system is running in the payload on a payload computer, receiving not only updated navigation positions (in the form of `NAV_DEPTH`, `NAV_X`, and `NAV_Y`), but also a "heartbeat" signal each time a new GPS position has been received (`GPS_RECEIVED`). This heartbeat signal may be enough to indicate to the helm and mission configuration that the objective of the surface excursion has been achieved.

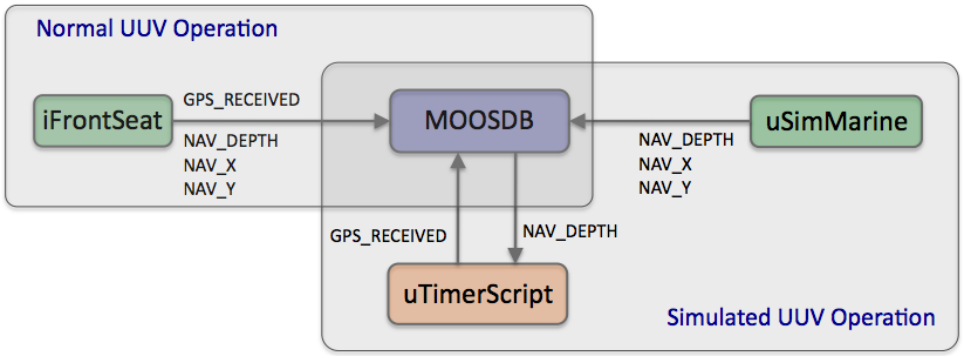


Figure 85: **Simulating a GPS Acknowledgment:** In a physical operation of the vehicle, the navigation solution and a GPS_UPDATE_RECEIVED heartbeat are received from the main vehicle (front-seat) computer via a MOOS module acting as an interface to the front-seat computer. In simulation, the navigation solution is provided by the simulator without any GPS_UPDATE_RECEIVED heartbeat. This element of simulation may be provided with uTimerScript configured to post the heartbeat, conditioned on the NAV_DEPTH information and a user-specified start delay to simulate GPS acquisition delay.

In simulation, however, the simulator only produces a steady stream of navigation updates with no regard to a simulated GPS unit. At this point there are three choices: (a) modify the simulator to fake GPS heartbeats and satellite delay, (b) write a separate simple MOOS application to do the same simulation. The drawback of the former is that one may not want to branch a new version of the simulator, or even introduce this new complexity to the simulator. The drawback of the latter is that, if one wants to propagate this functionality to other users, this requires distribution and version control of a new MOOS application.

A third and perhaps preferable option (c) is to write a short script for `uTimerScript` simulating the desired GPS characteristics. This achieves the objectives without modifying or introducing new source code. The below script in Listing 43 gets the job done.

Listing 43 - A uTimerScript configuration for simulating aspects of a GPS unit.

```
1  //------------------------------------------------------------
2  // uTimerScript configuration block
3
4  ProcessConfig = uTimerScript
5  {
6    AppTick   = 4
7    CommsTick = 4
8
9    paused      = false
10   reset_max   = unlimited
11   reset_time  = end
12   condition   = NAV_DEPTH < 0.2
13   upon_awake  = restart
14   delay_start = 20:120
15   script_name = GPS_SCRIPT
16
17   event  = var=GPS_UPDATE_RECEIVED, val="RCVD_$[COUNT]", time=0:1
18 }
```

This script posts a GPS_UPDATE_RECEIVED heartbeat message roughly once every second, based on the event time "time=0:1" on line 17. The value of this message will be unique on each posting due to the $[COUNT] macro in the value component. See Section 14.4.1 for more on macros. The script is configured to restart each time it awakes (line 13), defined by meeting the condition of (NAV_DEPTH < 0.2) which is a proxy for the vehicle being at the surface. The delay_start simulates the time needed for the GPS unit to reacquire satellite signals and is configured to be somewhere in the range of 20 to 120 seconds (line 14). Once the script gets past the start delay, the script is a single event (line 17) that repeats indefinitely since reset_max is set to unlimited and reset_time is set to end in lines 10 and 11. This script is used in the IvP Helm example simulation mission labeled "s4_delta" illustrating the PeriodicSurface helm behavior.

14.9.2 A Script as a Proxy for Simulating Random Wind Gusts

Simulating wind gusts, or in general, somewhat random external periodic drift effects on a vehicle, are useful for testing the robustness of certain autonomy algorithms. Often they don't need to be grounded in very realistic models of the environment to be useful, and here we show how a script can be used simulate such drift effects in conjunction with the uSimMarine application.

The uSimMarine application is a simple simulator that produces a stream of navigation information, NAV_X, NAV_Y, NAV_SPEED, NAV_DEPTH, and NAV_HEADING (Figure 86), based on the vehicle's last known position and trajectory, and currently observed values for actuator variables. The simulator also stores local state variables reflecting the current external drift in the x-y plane, by default zero. An external drift may be specified in terms of a drift vector, in absolute terms with the variable USM_DRIFT_VECTOR, or in relative terms with the variables USM_DRIFT_VECTOR_ADD.

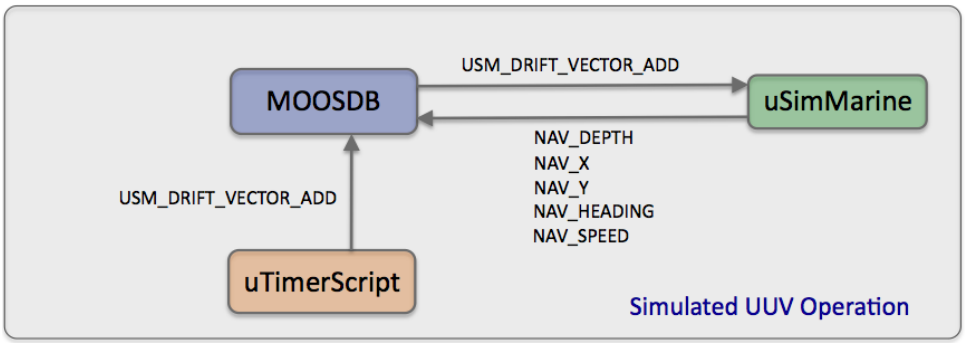


Figure 86: **Simulated Wind Gusts:** The uTimerScript application may be configured to post periodic sequences of external drift values, used by the uSimMarine application to simulate wind gust effects on its simulated vehicle.

The script in Listing 44 makes use of the uSimMarine interface by posting periodic drift vectors. It simulates a wind gust with a sequence of five posts to increase a drift vector (lines 18-22), and complementary sequence of five posts to decrease the drift vector (lines 24-28) for a net drift of zero at the end of each script execution.

Listing 44 - A uTimerScript configuration for simulating simple wind gusts.

```
0  //------------------------------------------
```

```
1  // uTimerScript configuration  block
2
3  ProcessConfig = uTimerScript
4  {
5    AppTick   = 2
6    CommsTick = 2
7
8    paused        = false
9    reset_max     = unlimited
10   reset_time    = end
11   delay_reset   = 10:60
12   time_warp     = 0.25:2.0
13   script_name   = WIND
14   script_atomic = true
15
16   randvar = varname=ANG, min=0,   max=359, key=at_reset
17   randvar = varname=MAG, min=0.5, max=1.5, key=at_reset
18
19   event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*0.2}", time=0
20   event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*0.2}", time=2
21   event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*0.2}", time=4
22   event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*0.2}", time=6
23   event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*0.2}", time=8
24
25   event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*-0.2}", time=10
26   event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*-0.2}", time=12
27   event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*-0.2}", time=14
28   event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*-0.2}", time=16
29   event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*-0.2}", time=18
30 }
```

The drift *angle* is chosen randomly in the range of $[0, 359]$ by use of the random variable macro `$[ANG]` defined on line 16. The peak *magnitude* of the drift vector is chosen randomly in the range of $[0.5, 1.5]$ with the random variable macro `$[MAG]` defined on line 17. Note that these two macros have their random values reset each time the script begins, by using the `key=at_reset` option, to ensure a stream of wind gusts of varying angles and magnitudes.

The duration of each gust sequence also varies between each script execution. The default duration is about 20 seconds, given the timestamps of 0 to 18 seconds in lines 19-29. The `time_warp` option on line 12 affects the duration with a random value chosen from the interval of $[0.25, 2.0]$. A time warp of 0.25 results in a gust sequence lasting about 80 seconds, and 2.0 results in a gust of about 10 seconds. The time between gust sequences is chosen randomly in the interval $[10, 60]$ by use of the `delay_restart` parameter on line 11. Used in conjunction with the `time_warp` parameter, the interval for possible observed delays between gusts is $[5, 240]$. The `reset_time` parameter set to `end`, on line 10 is used to ensure that the script posts all drift vectors to avoid any accumulated drifts over time. The `reset_max` parameter is set to `"unlimited"` to ensure the script runs indefinitely.

15 pBasicContactMgr: Managing Platform Contacts

15.1 Overview

The `pBasicContactMgr` application deals with information about other known vehicles in its vicinity. It is not a sensor application, but rather handles incoming "contact reports" which may represent information received by the vehicle over a communications link, or may be the result of on-board sensor processing. By default the `pBasicContactMgr` posts to the MOOSDB summary reports about known contacts, but it also may be configured to post alerts, i.e., MOOS variables, with select content about one or more of the contacts.

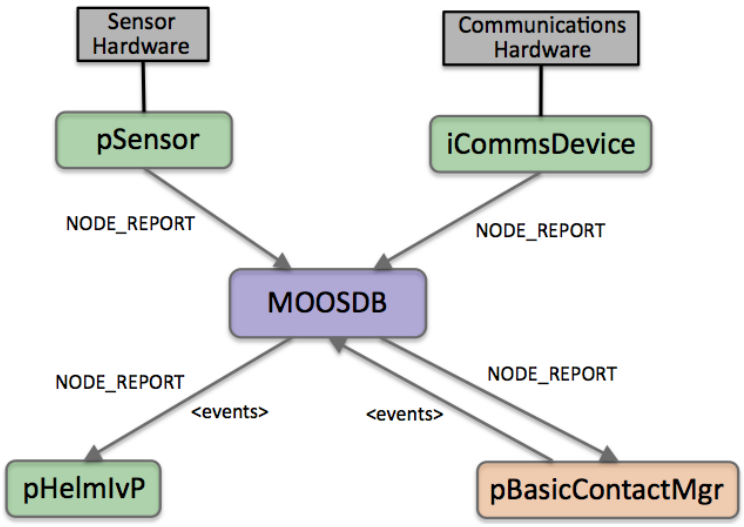


Figure 87: **The pBasicContactMgr Application:** The `pBasicContactMgr` utility receives `NODE_REPORT` information from other MOOS applications and manages a list of unique contact records. It may post additional user-configurable alerts to the MOOSDB based on the contact information and user-configurable conditions. The source of contact information may be external (via communications) or internal (via on-board sensor processing). The pSensor and iCommsDevice modules shown here are fictional applications meant to convey these two sources of information abstractly.

The `pBasicContactMgr` application is partly designed with simultaneous usage of the IvP Helm in mind. The alerts posted by `pBasicContactMgr` may be configured to trigger the dynamic spawning of behaviors in the helm, such as collision-avoidance behaviors. The `pBasicContactMgr` application does not perform sensor fusion, and does not reason about or post information regarding the confidence it has in the reported contact position relative to ground truth. These may be features added in the future, or perhaps may be features of an alternative contact manager application developed by a third party source.

15.2 Using pBasicContactMgr

The operation of `pBasicContactMgr` consists of posting user-configured alerts, and the posting of several MOOS variables, the `CONTACTS_*` variables, indicating the status of the contact manager.

15.2.1 Contact Alert Messages

Alert messages are used to alert other MOOS applications when a contact has been detected within a certain range of ownship. Multiple alert types may be configured, each keyed on the *alert id*. A single alert type may be defined over several lines, where each line contains the alert id of the alert type being configured. Alerts are configured in the mission file with the `alert` parameter as follows:

```
alert = id=<alert-id>, var=<MOOSVar>,              pattern=<string>
alert = id=<alert-id>, alert_range=<distance>,     cpa_range=<distance>
alert = id=<alert-id>, alert_range_color=<color>, cpa_range_color=<color>
```

The `var=<MOOSVar>` component indicates the MOOS variable posted for the given alert. The `pattern=<string>` component may be any string with any, none, or all of the following macros available for expansion:

`$[VNAME]`:	The name of the contact.
`$[X]`:	The position of the contact in local x coordinates.
`$[Y]`:	The position of the contact in local y coordinates.
`$[LAT]`:	The latitude position of the contact in earth coordinates.
`$[LON]`:	The longitude position of the contact in earth coordinates.
`$[HDG]`:	The reported heading of the contact.
`$[SPD]`:	The reported speed of the contact.
`$[DEP]`:	The reported depth of the contact.
`$[VTYPE]`:	The reported vessel type of the contact.
`$[UTIME]`:	The UTC time of the last report for the contact.

If the right-hand side of the `pattern=<string>` component contains its own parsing separators, it is recommended that the entire `<alert-pattern>` string is put within double quotes to ensure proper parsing, as in Line 11 in Listing 47.

The `alert_range=<distance>` component represents a threshold range to a contact, in meters. When a contact moves within this range, an alert will be generated. If this distance is left unspecified, a default value will be used. The default value for all alert types is 1000 meters. This fallback default alert range may be changed with the configuration parameter `default_alert_range`.

The `cpa_range=<distance>` component also represents a threshold range, in meters. Typically this `cpa_range` is greater than the `alert_range` parameter. When a contact is noted to be within the `cpa_range`, the contact and ownship trajectories are considered, to calculate the closest point of approach (CPA). If the CPA range is determined to be within the `alert_range`, an alert is generated, even if the present range between ownship and contact is outside the `alert_range`.

The `alert_range_color=<color>` component indicates a desired color for rendering the `alert_range` circle. Rendering is done by `pBasicContactMgr` by posting to the variable `VIEW_CIRCLE`, typically handled by `pMarineViewer`. The default value is `"gray70"`. The `cpa_range_color=<color>` component similarly indicates a desired color for rendering the `cpa_range` circle. The default value is `"gray30"`. See Appendix C for more on colors.

The posting of rendering circles by `pBasicContactMgr` may be disabled in one of three ways. First, they may be disabled outright for all alerts all the time by setting `display_radii=false` in the MOOS configuration block. Second, they may be turned off by posting `BCM_DISPLAY_RADII=false` to the MOOSDB at any time. This hook may be configured into a button or action-pull-down menu item in `pMarineViewer` for example. This will override any static setting in the MOOS configuration block. The third method for disabling the circles is to specify the color `"invisible"` for any one of the alert ranges. This value is interpreted at least in `pMarineViewer` as in indication that it is not to be drawn. The latter method provides a finer-grained control of rendering some circles but not others.

See lines 10-13 in Listing 47 for an example alert configuration.

15.2.2 Contact Alert Triggers

Alerts are triggered for all contacts based on range between ownship and the reported contact position. It is assumed that each incoming contact report minimally contains the contact's name and present position. An alert will be triggered if the current range to the contact falls within the distance given by `alert_range`, as in Contact-A in Figure 88.

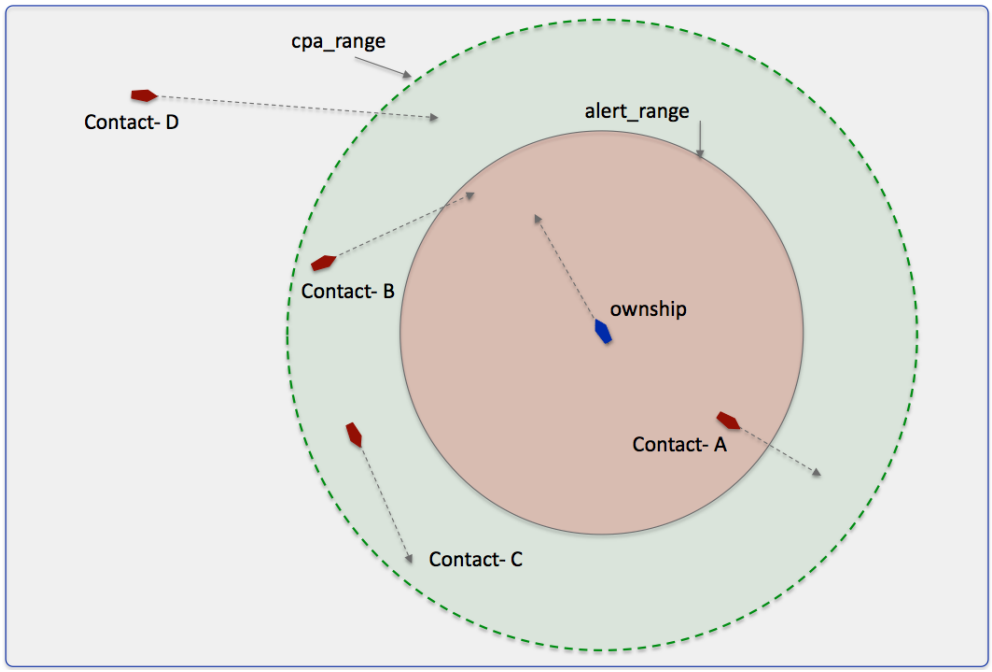


Figure 88: **Alert Triggers in pBasicContactMgr:** An alert may be triggered by pBasicContactMgr if the contact is within the `alert_range`, as with Contact-A. It may also be triggered if the contact is within the `cpa_range`, and the contact's CPA distance is within the `alert_range`, as with Contact-B. Contact-C shown here would not trigger an alert since its CPA distance is its current range and is not within the `alert_range`. Contact-D also would not trigger an alert despite the fact that its CPA with ownship is apparently small, since its current absolute range is greater than `cpa_range`.

The contact manager may also be configured with a second trigger criteria consisting of another

range to contact. The cpa_range may be set individually for a given user defined alert, but also has a default value which may be set in the configuration file:

```
alert_cpa_range = <distance>
```

The cpa_range is typically larger than the alert_range. (Its influence is effectively disabled when or if it is set to be equal to or less than the alert_range.) When a contact is outside the alert_range, but within the cpa_range, as with Contact-B in Figure 88, the closest point of approach (CPA) between the contact and ownship is calculated given their presently-known position and trajectories. If the CPA distance falls below the alert_range value, an alert is triggered.

15.2.3 Contact Alert Record Keeping

The contact manager keeps a record of all known contacts for which it has received a report. This list is posted in the MOOS variable CONTACTS_LIST, in a comma-separated string such as:

```
CONTACTS_LIST = "delta,gus,charlie,henry"
```

Once an alert is generated for a contact it is put on the *alerted* list and this subset of all contacts is posted in the MOOS variable CONTACTS_ALERTED. Each entry in the list names a vehicle and alert id separated by a comma, such as:

```
CONTACTS_ALERTED = "(delta,avd)(charlie,avd)"
```

Likewise, those contacts for which no alert has been generated are in the *unalerted* list and this is reflected in the MOOS variable CONTACTS_UNALERTED. Again, each entry is comprised of both a vehicle name and alert id separated by a comma.

```
CONTACTS_UNALERTED = "(gus,avd)(henry,avd)"
```

Contact records are not maintained indefinitely and eventually are *retired* from the records after some period of time during which no new reports are received for that contact. The period of time is given by the contact_max_age configuration parameter. The list of retired contacts is posted in the MOOS variable CONTACTS_RETIRED:

```
CONTACTS_RETIRED = "bravo,foxtrot,kilroy"
```

A contact recap of all non-retired contacts is also posted in the MOOS variable CONTACTS_RECAP:

```
CONTACTS_RECAP = "name=ike,age=11.3,range=193.1 # name=gus,age=0.7,range=48.2 # \
                 name=charlie,age=1.9,range=73.1 # name=henry,age=4.0,range=18.2"
```

Note: Each of these five MOOS variables is published only when its contents differ from its previous posting.

15.2.4 Contact Resolution

An alert is generated by the contact manager for a given contact *once*, when the alert trigger criteria is first met. In the iteration when the criteria is met, the contact is moved from the *unalerted* list to the *alerted* list, the alert is posted to the MOOSDB, and no further alerts are posted despite

any future calculations of the trigger criteria. One exception to this is when the pBasicContactMgr receives notice that a contact has been resolved, through the MOOS variable CONTACT_RESOLVED. When a contact is resolved, it is moved from the alerted list back on to the un-alerted list.

15.3 Deferring to Earth Coordinates over Local Coordinates

Incoming node reports contain the position information of the contact and may be specified in either local x-y coordinates, or earth latitude longitude coordinates or both. By default pBasicContactMgr uses the local coordinates for calculations and the earth coordinates are merely redundant. It may instead be configured, with the contact_local_coords parameter, to have its local coordinates set from the earth coordinates if the local coordinates are missing:

```
contact_local_coords = lazy_lat_lon
```

It may also be configured to always use the earth coordinates, even if the local coordinates are set:

```
contact_local_coords = force_lat_lon
```

The default setting is verbatim, meaning no action is taken to convert coordinates. If either of the other two above settings are used, the latitude and longitude coordinates of the local datum, or (0,0) point must be specified in the MOOS mission file, with LatOrigin and LongOrigin configuration parameters. (They are typically present in all mission files anyway.)

15.4 Usage of the pBasicContactMgr with the IvP Helm

The IvP helm may used in conjunction with the contact manager to coordinate the dynamic spawning of certain helm behaviors where the instance of the behavior is dedicated to a helm objective associated with a particular contact. For example, a collision avoidance behavior, or a behavior for maintaining a relative position to a contact for achieving sensing objectives, would be examples of such behaviors. One may want to arrange for a new behavior to be spawned as the contact becomes known. The helm needs a cue in the form of a MOOS variable posting to trigger a new behavior spawning, and this is easily arranged with the alerts in the pBasicContactMgr.

On the flip-side of a new behavior spawning, a behavior may eventually declare itself completed and remove itself from the helm. The conditions leading to completion are defined within the behavior implementation and configuration. No cues external to the helm are required to make that happen. However, once an alert has been generated by the contact manager for a particular contact, it is not generated again, unless it receives a message that the contact has been resolved. Therefore, if the helm wishes to received future alerts related to a contact for which it has received an alert in the past, it must declare the contact *resolved* to the contact manager as discussed in Section 15.2.4. This would be important, for example, in the following scenario: (a) a collision avoidance behavior is spawned for a new contact that has come within range, (b) the behavior completes and is removed from the helm, presumably because the contact has slipped safely out of range, (c) the contact or ownship turns such that a collision avoidance behavior is once again needed for the same contact.

An example mission is available for showing the use of the contact manager and its coordination with the helm to spawn behaviors for collision avoidance. This mission is `m2_berta` and is described in the IvP Helm documentation. In this mission two vehicles are configured to repeatedly go in and out of collision avoidance range, and the contact manager repeatedly posts alerts that result in the spawning of a collision avoidance behavior in the helm. Each time the vehicle goes out of range, the behavior completes and dies off from the helm and is declared to the contact manager to be resolved.

15.5 Terminal and AppCast Output

The status of the contact manager may be monitored from from an open console window where pBasicContactMgr is launched. Example output is shown below in Listing 45.

Listing 45 - Example pBasicContactMgr terminal and appcast output.

```
 1  ============================================================================
 2  pBasicContactMgr gilda                                          0/0 (379)
 3  ============================================================================
 4  Alert Configurations (1):
 5  ---------------------
 6  Alert ID = avd
 7    VARNAME   = CONTACT_INFO
 8    PATTERN   = name=$[VNAME]#contact=$[VNAME]
 9    RANGE     = 40, green
10    CPA_RANGE = 45, invisible
11
12  Alert Status Summary:
13  ---------------------
14         List: henry
15      Alerted:
16    UnAlerted: (henry,avd)
17      Retired:
18        Recap: vname=henry,range=136.55,age=2.05
19
20  Contact Status Summary:
21  ----------------------
22  Contact     Range     Alerts     Alerts     Alerts
23                        Total      Active     Resolved
24  -------     -----     ------     ------     --------
25  henry       136.6     1          0          1
26
27
28  Recent Events (3):
29  [159.35]: Resolved: (henry,all_alerts)
30  [159.35]: TryResolve: (henry,all_alerts)
31  [104.53]: CONTACT_INFO=name=henry#contact=henry
```

On line 2, the `"0/0"` indicates there were no configuration warnings and no run-time warnings (thus far). The `"(379)"` represents the iteration counter of `pBasicContactMgr`. In lines 4-10, the alerts configured by the user in the MOOS configuration block are shown. If multiple alerts types are configured, they would each be listed here separated by their alert id.

In lines 12-18, the record-keeping status of the contact manager is output. These five lines are equivalent to the content of the CONTACTS_* variables described in Section 15.2.3. In lines 20-25, the status and alert history for each known contact is shown. Finally, in lines 28, a limited list of recent events is shown. Typically an event is either an alert generated or an alert resolved. The alert resolution is split into two events, the alert resolution attempt, and the actual resolution. This may help draw the user's attention if an alert is attempted but failed.

15.6 Configuration Parameters for pBasicContactMgr

The following parameters are defined for pBasicContactMgr. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so.

Listing 15.46: Configuration Parameters for pBasicContactMgr.

alert:	A description of a single alert. Section 15.2.1.
contact_local_coords:	Determines if the local coordinates of incoming node reports are filled by translated latitude longitude coordinates. Legal values: verbatim, lazy_lat_lon, force_lat_lon. The default is verbatim, meaning no translation action is taken.
default_alert_range:	The range to a contact, in meters, within which an alert is posted. Legal values: any positive number. The default is 1000. Section 15.2.1.
default_cpa_range:	The range to a contact, in meters, within which an alert is posted if the closest point of approach (CPA) falls within this range. Legal values: any positive number. The default is 1000. Section 15.2.1.
default_alert_range_color:	The default color for rendering the alert range radius. Legal values: any color in Appendix C. The default is gray70. Section 15.2.1.
default_cpa_range_color:	The default color for rendering the cpa range radius. Legal values: any color in Appendix C. The default is gray30. Section 15.2.1.
contact_max_age:	Seconds between reports before a contact is dropped from the list. Legal values: any non-negative number. The default is 3600. Section 15.2.3.
display_radii:	If true, the two alert ranges are posted as viewable circles. Legal values: true, false. The default is false.

15.6.1 An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command-line:

```
$ pBasicContactMgr --example or -e
```

This will show the output shown in Listing 47 below.

Listing 47 - Example configuration of the pBasicContactMgr application.

```
 0  ============================================================
 1  pBasicContactMgr Example MOOS Configuration
 2  ============================================================
 3
 4  ProcessConfig = pBasicContactMgr
 5  {
 6    AppTick   = 4
 7    CommsTick = 4
 8
 9    // Alert configurations (one or more, keyed by id)
10    alert = id=avd, var=CONTACT_INFO
11    alert = id=avd, pattern="name=avd_$[VNAME] # contact=$[VNAME]"
12    alert = id=avd, alert_range=80, alert_range_color=white
13    alert = id=avd, cpa_range=95, cpa_range_color=gray50
14
15    // Properties for all alerts
16    default_alert_range = 1000    // meters.   Range [0,inf)
17    default_cpa_range   = 1000    // meters.   Range [0,inf)
18
19    // Policy for retaining potential stale contacts
20    contact_max_age     = 3600    // seconds.  Range [0,inf)
21
22    // Configuring other output
23    display_radii        = false  // or {true}
24  }
```

15.7 Publications and Subscriptions for pBasicContactMgr

The interface for `pBasicContactMgr`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pBasicContactMgr --interface or -i
```

15.7.1 Variables Published by pBasicContactMgr

The primary output of pBasicContactMgr to the MOOSDB is the set of user-configured alerts. Other variables are published on each iteration where a change is detected on its value:

- `APPCAST`: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 15.5.
- `CONTACTS_LIST`: A comma-separated list of contacts.
- `CONTACTS_RECAP`: A comma-separated list of contact summaries.
- `CONTACTS_ALERTED`: A list of contacts for which alerts have been posted.
- `CONTACTS_UNALERTED`: A list of contacts for which alerts are pending, based on the range criteria.
- `CONTACTS_RETIRED`: A list of contacts removed due to the information staleness.
- `CONTACT_MGR_WARNING`: A warning message indicating possible mishandling of or missing data.

- VIEW_CIRCLE: A rendering of the alert ranges.

Some examples:

```
CONTACTS_LIST      = gus,joe,ken,kay
CONTACTS_ALERTED   = gus,kay
CONTACTS_UNALERTED = ken,joe
CONTACTS_RETIRED   = bravo,foxtrot,kilroy
CONTACTS_RECAP     = name=gus,age=7.3,range=13.1 # name=ken,age=0.7,range=48.1 # \
                     name=joe,age=1.9,range=73.1 # name=kay,age=4.0,range=18.2
```

15.7.2 Variables Subscribed for by pBasicContactMgr

The pBasicContactMgr application will subscribe for the following MOOS variables:

- APPCAST_REQ: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section **??**.
- BCM_DISPLAY_RADII: If false, no postings will be made for rendering the alert and cpa range circles.
- CONTACT_RESOLVED: A name of a contact that has been declared resolved, possibly with a particular alert specified.
- NAV_X: Present position of ownship in local x coordinates.
- NAV_Y: Present position of ownship in local y coordinates.
- NAV_HEADING: Present ownship heading in degrees.
- NAV_SPEED: Present ownship speed in meters per second.
- NODE_REPORT: A report about a known contact.

15.7.3 Command Line Usage of pBasicContactMgr

The pBasicContactMgr application is typically launched as a part of a batch of processes by pAntler, but may also be launched from the command line by the user. To see command-line options enter the following from the command-line:

```
$ pBasicContactMgr --help
```

This will show the output shown in Listing 48 below.

Listing 48 - Command line usage for the pBasicContactMgr *application.*

```
1  ============================================================
2  Usage: pBasicContactMgr file.moos [OPTIONS]
3  ============================================================
4
5  SYNOPSIS:
6  ------------------------------------
7    The contact manager deals with other known vehicles in its
8    vicinity. It handles incoming reports perhaps received via a
9    sensor application or over a communications link. Minimally
10   it posts summary reports to the MOOSDB, but may also be
```

```
11    configured to post alerts with user-configured content about
12    one or more of the contacts.
13
14  Options:
15    --alias=<ProcessName>
16        Launch pBasicContactMgr with the given process
17        name rather than pBasicContactMgr.
18    --example, -e
19        Display example MOOS configuration block.
20    --help, -h
21        Display this help message.
22    --interface, -i
23        Display MOOS publications and subscriptions.
24    --version,-v
25        Display the release version of pBasicContactMgr.
26
27  Note: If argv[2] does not otherwise match a known option,
28        then it will be interpreted as a run alias. This is
29        to support pAntler launching conventions.
```

A Use of Logic Expressions

Logic conditions are employed in both the `pHelmIvP` and `uTimerScript` applications, to condition certain activities based on the prescribed logic state of elements of the MOOSDB. The use of logic conditions in the helm is done in behavior file (`.bhv` file). For the `uTimerScript` application, logic conditions are used in the configuration block of the mission file (`.moos` file). The MOOS application using logic conditions maintains a local buffer representing a snapshot of the MOOSDB for variables involved in the logic expressions. The key relationships and steps are shown in Figure 89:

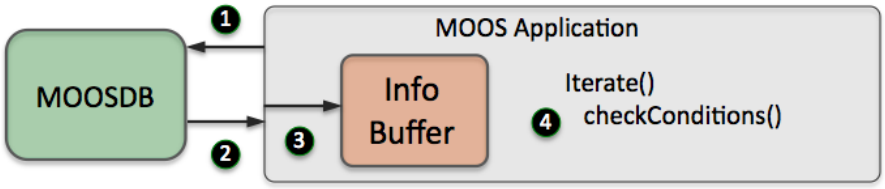


Figure 89: **Logic conditions in a MOOS application**: Step 1: the applications registers to the MOOSDB for any MOOS variables involved in the logic expressions. Step 2: The MOOS application reads incoming mail from the MOOSDB. Step 3: Any new mail results in an update to the information buffer. Step 4: Within the applications Iterate() method, the logic expressions are evaluated based on the contents of the information buffer.

The logic conditions are configured as follows:

```
condition = <logic-expression>
```

The parameter `condition` is case insensitive. When multiple conditions are specified, it is implied that the overall criteria for meeting conditions is the conjunction of all such conditions. In what remains below, the allowable syntax for `<logic-expression>` is described.

Simple Relational Expressions

Each logic expression is comprised of either Boolean operators (and, or, not) or relation operators ($\leq, <, \geq, >, =, !=$). All expressions have at least one relational expression, where the left-hand side of the expression is treated as a variable, and the right-hand side is a literal (either a string or numerical value). The literals are treated as a string value if quoted, or if the value is non-numerical. Some examples:

```
condition = (DEPLOY   = true)    // Example 1
condition = (QUALITY >= 75)      // Example 2
```

Variable names are case sensitive since MOOS variables in general are case sensitive. In matching string values of MOOS variables in Boolean conditions, the matching is *case insensitive*. If for example, in Example 1 above, the MOOS variable `DEPLOY` had the value `"TRUE"`, this would satisfy the condition. But if the MOOS variable `deploy` (lowercase is unconventional, but legal) had the value `"true"`, this would not satisfy Example 1.

Simple Logical Expressions with Two MOOS Variables

A relational expression generally involves a variable and a literal, and the form is simplified by insisting the variable is on the left and the literal on the right. A relational expression can also involve the comparison of two variables by surrounding the right-hand side with `$()`. For example:

```
condition = (REQUESTED_STATE != $(RUN_STATE))     // Example 3
```

The variable types need to match or the expression will evaluate to `false` regardless of the relation. The expression in Example 3 will evaluate to `false` if, for example, `REQUESTED_STATE="run"` and `RUN_STATE=7`, simply because they are of different type, and regardless of the relation being the inequality relation.

Complex Logic Expressions

Individual relational expressions can be combined with Boolean connectors into more complex expressions. Each component of a Boolean expression must be surrounded by a pair of parentheses. Some examples:

```
condition = (DEPLOY = true) or (QUALITY >= 75)              // Example 4

condition = (MSG != error) and !((K <= 10) or (w != 0))     // Example 5
```

A relational expression such as `(w != 0)` above is false if the variable `w` is undefined. In MOOS, this occurs if variable has yet to be published with a value by any MOOS client connected to the MOOSDB. A relational expression is also `false` if the variable in the expression is the wrong type, compared to the literal. For example `(w != 0)` in Example 5 would evaluate to `false` even if the variable `w` had the string value `"alpha"` which is clearly not equal to zero.

B Behavior Summaries

Parameter Summary for BHV_Waypoint

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
capture_radius	double	7	-	0	101
cycleflag	MOOSVAR=value	CYCLED=true	yes	-	103
lead	double	10	-	-1	102
lead_damper	double	2	-	-1	102
lead_on_start	Boolean string	TRUE	no	false	102
slip_radius	double	18	-	0	101
order	string	reverse	no	normal	101
point	string	20,5	yes	-	101
points, polygon	string	0,0:45,0:45,80:0,0	yes	-	101
post_suffix	string	IKE	yes	""	100
repeat	int	3	no	0	101
speed	double	1.2	-	0	100
visual_hints	string	edge_size=1	no	-	100

Table 31: Parameters for the BHV_Waypoint behavior.a

Variables posted by the BHV_Waypoint Behavior

The following variables may be posted by the BHV_Waypoint behavior in addition to any configured flags (with the runflag, idleflag, activeflag, inactiveflag, and cycleflag parameters).

- VIEW_POINT, VIEW_SEGLIST, WPT_INDEX, CYCLE_INDEX, WPT_STAT

The variable names used may be changed with the POST_MAPPING parameter. See page 82.

Example Behavior File Configuration for BHV_Waypoint

Listing B.1 - An example BHV_Waypoint configuration.

```
 0  Behavior = BHV_Waypoint
 1  {
 2    name       = waypt_survey
 3    priority   = 100
 4    updates    = WPT_SURVEY_UPDATES
 5    condition  = (DEPLOY == true) or (SURVEY == on))
 6    endflag    = SURVEY = COMPLETE
 7
 8            points = label,survey_points:-57,-60:-70,-109:-77,-144:-51
 9
10             speed = 3.0  // meters per second
11    capture_radius = 8.0  // meters
12         nm_radius = 16.5 // meters
13            repeat = 0    // number of iterations
14              lead = 10   // meters
15  }
```

Parameter Summary for BHV_OpRegion

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
polygon	string	0,0:45,0:45,80:0,80:0,0	-	-	107
max_depth	double	200	-	0	108
min_altitude	double	25	-	0	108
max_time	double	3600	-	0	107
trigger_entry_time	double	1.5	-	0	107
trigger_exit_time	double	2.4	-	0	107
visual_hints	string	edge_size=1	no	-	106

Table 32: Parameters for the BHV_OpRegion behavior.

Example Behavior File Configuration for BHV_OpRegion

Listing B.2 - An example BHV_OpRegion configuration.

```
 0  Behavior = BHV_OpRegion
 1  {
 2    name               = bhv_opregion
 3    polygon            = label,opregion : -57,-60 : -70,-109 : -77,-144
 4
 5    max_depth          = 50     // meters
 6    min_altitude       = 10     // meters
 7    max_time           = 3600   // seconds
 8    trigger_entry_time = 0.5    // seconds
 9    trigger_exit_time  = 1.0    // seconds
10 }
```

Parameter Summary for BHV_Loiter

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
acquire_dist	double	15	-	10	116
capture_radius	double	10	-	0	115
center_activate	Boolean	true	no	false	110
center_assign	string	present_position	no	-	110
clockwise	Boolean string	FALSE	no	true	116
polygon	string	0,0:45,0:45,80:0,80:0,0	yes	-	115
post_suffix	string	REGION-1	yes	""	116
speed	double	1.5	-	0	115
slip_radius	double [0,100]	25	-	0	115
spiral_factor	double	45	-	99	110
visual_hints	string	edge_size=1	no	-	110
xcenter_assign	double	20	no	-	110
ycenter_assign	double	25	no	-	110

Table 33: Parameters for the BHV_Loiter behavior.

Example Behavior File Configuration for BHV_Loiter

Listing B.3 - An example BHV_Loiter configuration.

```
 0  Behavior = BHV_Loiter
 1  {
 2    name     = loiter_alpha
 3    pwt      = 100
 4    duration = 3600 // One hour
 5    updates  = LOITER_ALPHA_UPDATES
 7
 8           polygon = radial:100,-100,80,12
 9             speed = 3.0
10    capture_radius = 8.0
11       slip_radius = 16.0
12         clockwise = true
13       aquire_dist = 25
14     center_assign = present_position
14  }
```

Parameter Summary for BHV_PeriodicSpeed

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
basewidth	double	0.5	-	0	117
initially_busy	Boolean string	true	no	false	117
peakwidth	double	0.2	-	0	117
period_busy	double	60	-	0	117
period_lazy	double	600	-	0	117
period_speed	double	0.8	-	0	117
reset_upon_running	Boolean string	false	no	true	117
summit_delta	double $[0, 100]$	0.5	-	25	117

Table 34: Parameters for the BHV_PeriodicSpeed behavior.

Example Behavior File Configuration for BHV_PeriodicSpeed

Listing B.3 - An example BHV_PeriodicSpeed configuration.

```
 0  Behavior = BHV_PeriodicSpeed
 1  {
 2    name     = periodic_speed
 3    priority = 500
 4
 5        period_length = 30    // Seconds
 6           period_gap = 120   // Seconds
 7        reset_on_idle = true  // The default
 7       initially_busy = false // The default
 7         period_speed = 0.5   // Meters/sec
 8            peakwidth = 0.3   // Meters/sec
 9            basewidth = 0.5   // Meters/sec
 9         summit_delta = 25    // The default
12  }
```

Parameter Summary for BHV_PeriodicSurface

Parameter	Argument Type	Example	Case-Sense	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
acomms_mark_variable	MOOSVAR	RANGE_RECEIVED	yes	-	120
ascent_grade	string	quasi	no	linear	120
ascent_speed	double	1.0	-	*	120
mark_variable	MOOSVAR	GPS_RECEIVED	yes	GPS_UPDATE_RECEIVED	120
max_time_at_surface	MOOSVAR	60	yes	300	120
period	double	60	-	300	120
zero_speed_depth	double	2.5	-	0	120

Table 35: Parameters for the BHV_PeriodicSurface behavior.

Example Behavior File Configuration for BHV_PeriodicSurface

Listing B.4 - An example BHV_PeriodicSurface configuration.

```
 0  Behavior = BHV_PeriodicSurface
 1  {
 2    name          = bhv_periodic_surface
 3    priority      = 500
 4    active_flag   = SURFACING, IN_PROGRESS
 5    inactive_flag = SURFACING, NO
 6
 7                  period = 3600     // seconds
 8            ascent_speed = 1.0      // meters per second
 9        zero_speed_depth = 2.5      // meters
10     max_time_at_surface = 120      // seconds
11            ascent_grade = linear
12    acomms_mark_variable = RANGE_RECEIVED
13           mark_variable = GPS_UPDATE_RECEIVED
14         status_variable = PERIODIC_PENDING_SURFACE
15  }
```

Parameter Summary for BHV_ConstantDepth

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
basewidth	double	15	-	100	121
depth	double	35	-	0	121
peakwidth	double	5	-	3	121
summitdelta	double	20	-	50	121

Table 36: Parameters for the BHV_ConstantDepth behavior.

Example Behavior File Configuration for BHV_ConstantDepth

Listing B.5 - An example BHV_ConstantDepth configuration.

```
 0  Behavior = BHV_ConstantDepth
 1  {
 2    // General Behavior Parameters
 3    name      = constant_depth_survey
 4    priority  = 100
 5    condition = AUTONOMY_MODE = SURVEY
 6    duration  = no-time-limit
 7    updates   = NEW_SURVEY_DEPTH
 8    nostarve  = NAV_DEPTH, 3.0
 9
10    // BHV_ConstantDepth Behavior Parameters
11          depth = 50      // meters
12      peakwidth = 5
13      basewidth = 10
14    summitdelta = 45
15  }
```

Parameter Summary for BHV_ConstantHeading

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
basewidth	double	175	-	170	122
heading	double	35	-	0	122
peakwidth	double	5	-	10	122
summitdelta	double	35	-	25	122

Table 37: Parameters for the BHV_ConstantHeading behavior.

Example Behavior File Configuration for BHV_ConstantHeading

Listing B.6 - An example BHV_ConstantHeading configuration.

```
 0  Behavior = BHV_ConstantHeading
 1  {
 2    name      = bhv_constant_heading
 3    priority  = 100
 4    duration  = 60
 5    condition = AUTONOMY_MODE = PID_TEST
 6    updates   = NEW_TEST_HEADING
 7    nostarve  = NAV_HEADING, 3.0
 8
 9        heading = 45  // degrees
10      peakwidth = 30
11      basewidth = 150
12    summitdelta = 25
13  }
```

Parameter Summary for BHV_ConstantSpeed

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
basewidth	double	0.6	-	2.0	123
speed	double	1.2	-	0.0	123
peakwidth	double	0.1	-	0.0	123
summitdelta	double	35	-	0	123

Table 38: Parameters for the BHV_ConstantSpeed behavior.

Example Behavior File Configuration for BHV_ConstantSpeed

Listing B.7 - An example BHV_ConstantSpeed configuration.

```
 0  Behavior = BHV_ConstantSpeed
 1  {
 2    name        = const_speed_bravo
 3    priority    = 100
 4    duration    = 60
 5    active_flag = BRAVO_SPEED_TEST = in-progress
 6    nostarve    = NAV_SPEED, 2.0
 7
 8      speed     = 1.8  // meters per second
 9      peakwidth = 0.3
10      basewidth = 1.0
11    summitdelta = 22
12  }
```

Parameter Summary for BHV_GoToDepth

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
depth, depths	string	50,10:40,60	yes	-	124
repeat	int	5	-	0	124
capture_delta	double	2	-	1	124
capture_flag	MOOSVAR	DEPTH_HIT	yes	-	124

Table 39: Parameters for the BHV_GoToDepth behavior.

Example Behavior File Configuration for BHV_GoToDepth

Listing B.8 - An example BHV_GoToDepth configuration.

```
 0  Behavior = BHV_GoToDepth
 1  {
 2    name      = goto_depth_set_alpha
 3    priority  = 100
 4    condition = DEPLOY == true
 9    endflag   = GOTO_DEPTH_ALPHA = DONE
 5
 6          depths = 15,30: 30,30: 45,60: 15,30
 7    capture_delta = 1  // meters
 8     capture_flag = DEPTH_LEVELS_ACHIEVED
 9          repeat = 4
10  }
```

Parameter Summary for BHV_MemoryTurnLimit

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
memory_time	double	60	-	-1	126
turn_range	double	45	-	-1	126

Table 40: Parameters for the BHV_MemoryTurnLimit behavior.

Example Behavior File Configuration for BHV_MemoryTurnLimit

Listing B.9 - An example BHV_MemoryTurnLimit configuration.

```
0  Behavior = BHV_MemoryTurnLimit
1  {
2    name      = memturnlimit
3    priority  = 1000
4
5    memory_time = 60  // seconds
6     turn_range = 35  // degrees
7  }
```

Parameter Summary for BHV_StationKeep

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
station_pt, point	string	50,75	yes	0,0	129
center_activate	Boolean string	TRUE	no	false	129
inner_radius	double	10	-	4	129
outer_radius	double	25	-	15	129
outer_speed	double	1.8	-	1.2	129
transit_speed	double	1.9	-	2.5	129
passive_station_radius	double	200	-	0	130
passive_station_variable	MOOSVAR	PSK_MODE_CHARLIE	-	PSKEEP_MODE	130

Table 41: Parameters for the BHV_StationKeep behavior.

Example Behavior File Configuration for BHV_StationKeep

Listing B.10 - An example BHV_StationKeep configuration.

```
 0  Behavior = BHV_StationKeep
 1  {
 2    name       = bhv_station_keep
 3    priority   = 100
 4    condition  = (ON_STATION=true) and (RETURN=false)
 5    updates    = STATION_UPDATES
 6
 7                  station_pt = 200,-150
 8              center_activate = true
 9                 inner_radius = 10
10                 outer_radius = 40
11                  outer_speed = 0.8
12                transit_speed = 1.8
13      passive_station_radius = 400          // meters
14    passive_station_variable = PSKEEP_MODE  // the default
15  }
```

Parameter Summary for BHV_Timer

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
No additional parameters for this behavior					

Table 42: Parameters for the BHV_Timer behavior.

Example Behavior File Configuration for BHV_Timer

Listing B.11 - An example BHV_Timer configuration.

```
0  Behavior = BHV_Timer
1  {
2    name      = bhv_timer_a
3    duration  = 60             // seconds
4    condition = loiter = alpha
5    end_flag  = loiter = beta
6  }
```

Parameter Summary for BHV_AvoidCollision

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
contact	string	Alliance	yes	-	137
on_no_contact_ok	boolean	true	no	true	138
extrapolate	boolean	true	no	false	137
decay	double,double	10, 30	-	0,0	137
bearing_lines	string		no	-	141
pwt_grade	string	quadratic	no	linear	141
completed_dist	double	200	-	500	141
pwt_inner_dist	double	50	-	50	141
pwt_outer_dist	double	200	-	200	141
max_util_cpa_dist	double	100	-	75	141
min_util_cpa_dist	double	10	-	10	141

Table 43: Parameters for the BHV_AvoidCollision behavior.

Example Behavior File Configuration for BHV_AvoidCollision

Listing B.12 - An example BHV_AvoidCollision configuration.

```
 0  Behavior = BHV_AvoidCollision
 1  {
 2    name      = avoid_collision_alpha
 3    pwt       = 100
 4    condition = AVOIDANCE_MODE != INACTIVE
 5
 6                contact = alpha
 7       on_no_contact_ok = true
 8            extrapolate = true
 9                  decay = 30,60
10
11           pwt_outer_dist = 150
12           pwt_inner_dist = 75
13        min_util_cpa_dist = 15
14        max_util_cpa_dist = 80
15              pwt_grade = linear
16     bearing_line_config = white:0, green:0.65, yellow:0.8, red:1.016
14  }
```

Parameter Summary for BHV_CutRange

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
contact	string	Alliance	yes	-	137
on_no_contact_ok	boolean	true	no	true	138
extrapolate	boolean	true	no	false	137
decay	double,double	10, 30	-	0,0	137
dist_priority_interval	double,double	40,100	-	0,0	145
time_on_leg	double	60	-	15	145
give_up_range	double	500	-	0	145
patience	double	50	-	0	145

Table 44: Parameters for the BHV_CutRange behavior.

Example Behavior File Configuration for BHV_CutRange

Listing B.13 - An example BHV_CutRange configuration.

```
0  Behavior = BHV_CutRange
1  {
2    name         = bhv_cutrange
3    pwt          = 100
4    contact      = zulu
5
5    dist_priority_interval = 25,100
6               time_on_leg = 60
7             give_up_range = 400
8                  patience = 75
9  }
```

Parameter Summary for BHV_Shadow

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
contact	string	Alliance	yes	-	137
on_no_contact_ok	boolean	true	no	true	138
extrapolate	boolean	true	no	false	137
decay	double,double	10, 30	-	0,0	137
max_range	double	100	-	0	145
heading_peakwidth	double	10	-	20	145
heading_basewidth	double	170	-	160	145
speed_peakwidth	double	0.3	-	0.1	145
speed_basewidth	double	0.5	-	2.0	145

Table 45: Parameters for the BHV_Shadow behavior.

Example Behavior File Configuration for BHV_Shadow

Listing B.14 - An example BHV_Shadow configuration.

```
 0  Behavior = BHV_Shadow
 1  {
 2    name      = bhv_shadow
 3    pwt       = 100
 4    contact   = delta
 5
 6            max_range = 200
 7    heading_peakwidth = 10
 8    heading_basewidth = 170
 9      speed_peakwidth = 10
10      speed_basewidth = 170
11  }
```

Parameter Summary for BHV_Trail

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	loiter-west-zone	yes	-	81
duration	double	600	-	-1	85
duration_status	MOOSVAR	loiter_remaining	yes	-	85
priority, pwt	double	100	-	100	81
runflag	MOOSVAR=value	LOITERING = maybe	yes	-	73
endflag	MOOSVAR=value	LOITERING = done	yes	-	73
activeflag	MOOSVAR=value	LOITERING = yes	yes	-	73
inactiveflag	MOOSVAR=value	LOITERING = off	yes	-	73
idleflag	MOOSVAR=value	LOITERING = no	yes	-	73
nostarve	MOOSVAR,double	INFO,60	yes	-	86
perpetual	Boolean string	false	no	false	85
post_mapping	MOOSVAR,MOOSVAR	FOO,BAR	yes	-	82
updates	MOOSVAR	LOITER_INFO	yes	-	84
condition	Logic Expression	QUALITY <= 7	yes	-	72
contact	string	Alliance	yes	-	137
on_no_contact_ok	boolean	true	no	true	138
extrapolate	boolean	true	no	false	137
decay	double,double	10, 30	-	0,0	137
trail_range	double	20	-	50	145
trail_angle	double	270	-	180	145
trail_angle_type	string	absolute	no	relative	145
radius	double	8	-	5	145
nm_radius	double	20	-	20	145
max_range	double	50	-	0	145

Table 46: Parameters for the BHV_Trail behavior.

Example Behavior File Configuration for BHV_Trail

Listing B.15 - An example BHV_Trail configuration.

```
 0  Behavior = BHV_Trail
 1  {
 2    name             = bhv_trail
 3    priority         = 100
 5
 5    contact          = delta
 6    extrapolate      = true
 7    on_no_contact_ok = true
 8    decay            = 20,60    // seconds
 9
10         trail_range = 50       // meters
11         trail_angle = 185      // degrees
12    trail_angle_type = relative
13               radius = 10      // meters
14           nm_radius = 30       // meters
15           max_range = 300      // meters
16  }
```

C Colors

Below are the colors used by IvP utilities that use colors. Colors are case insensitive. A color may be specified by the string as shown, or with the '_' character as a separator. Or the color may be specified with its hexadecimal or floating point form. For example the following are equivalent: "darkblue", "DarkBlue", "dark_blue", "hex:00,00,8b", and "0,0,0.545". In the latter two styles, the '%', '$', or '#' characters may also be used as a delimiter instead of the comma if it helps when embedding the color specification in a larger string that uses its own delimeters. Mixed delimeters are not supported however.

antiquewhite, (fa,eb,d7)
aqua (00,ff,ff)
aquamarine (7f,ff,d4)
azure (f0,ff,ff)
beige (f5,f5,dc)
bisque (ff,e4,c4)
black (00,00,00)
blanchedalmond(ff,eb,cd)
blue (00,00,ff)
blueviolet (8a,2b,e2)
brown (a5,2a,2a)
burlywood (de,b8,87)
cadetblue (5f,9e,a0)
chartreuse (7f,ff,00)
chocolate (d2,69,1e)
coral (ff,7f,50)
cornsilk (ff,f8,dc)
cornflowerblue(64,95,ed)
crimson (de,14,3c)
cyan (00,ff,ff)
darkblue (00,00,8b)
darkcyan (00,8b,8b)
darkgoldenrod (b8,86,0b)
darkgray (a9,a9,a9)
darkgreen (00,64,00)
darkkhaki (bd,b7,6b)
darkmagenta (8b,00,8b)
darkolivegreen(55,6b,2f)
darkorange (ff,8c,00)
darkorchid (99,32,cc)
darkred (8b,00,00)
darksalmon (e9,96,7a)
darkseagreen (8f,bc,8f)
darkslateblue (48,3d,8b)
darkslategray (2f,4f,4f)
darkturquoise (00,ce,d1)
darkviolet (94,00,d3)
deeppink (ff,14,93)
deepskyblue (00,bf,ff)
dimgray (69,69,69)
dodgerblue (1e,90,ff)
firenrick (b2,22,22)
floralwhite (ff,fa,f0)
forestgreen (22,8b,22)
fuchsia (ff,00,ff)
gainsboro (dc,dc,dc)
ghostwhite (f8,f8,ff)
gold (ff,d7,00)
goldenrod (da,a5,20)
gray (80,80,80)
green (00,80,00)
greenyellow (ad,ff,2f)
honeydew (f0,ff,f0)
hotpink (ff,69,b4)
indianred (cd,5c,5c)
indigo (4b,00,82)
ivory (ff,ff,f0)
khaki (f0,e6,8c)
lavender (e6,e6,fa)
lavenderblush (ff,f0,f5)
lawngreen (7c,fc,00)
lemonchiffon (ff,fa,cd)
lightblue (ad,d8,e6)
lightcoral (f0,80,80)
lightcyan (e0,ff,ff)
lightgoldenrod(fa,fa,d2)
lightgray (d3,d3,d3)
lightgreen (90,ee,90)

lightpink (ff,b6,c1)
lightsalmon (ff,a0,7a)
lightseagreen (20,b2,aa)
lightskyblue (87,ce,fa)
lightslategray(77,88,99)
lightsteelblue(b0,c4,de)
lightyellow (ff,ff,e0)
lime (00,ff,00)
limegreen (32,cd,32)
linen (fa,f0,e6)
magenta (ff,00,ff)
maroon (80,00,00)
mediumblue (00,00,cd)
mediumorchid (ba,55,d3)
mediumseagreen(3c,b3,71)
mediumslateblue(7b,68,ee)
mediumspringgreen(00,fa,9a)
mediumturquoise(48,d1,cc)
mediumvioletred(c7,15,85)
midnightblue (19,19,70)
mintcream (f5,ff,fa)
mistyrose (ff,e4,e1)
moccasin (ff,e4,b5)
navajowhite (ff,de,ad)
navy (00,00,80)
oldlace (fd,f5,e6)
olive (80,80,00)
olivedrab (6b,8e,23)
orange (ff,a5,00)
orangered (ff,45,00)
orchid (da,70,d6)
palegreen (98,fb,98)
paleturquoise (af,ee,ee)
palevioletred (db,70,93)
papayawhip (ff,ef,d5)
peachpuff (ff,da,b9)
pelegoldenrod (ee,e8,aa)
peru (cd,85,3f)
pink (ff,c0,cb)
plum (dd,a0,dd)
powderblue (b0,e0,e6)
purple (80,00,80)
red (ff,00,00)
rosybrown (bc,8f,8f)
royalblue (41,69,e1)
saddlebrowm (8b,45,13)
salmon (fa,80,72)
sandybrown (f4,a4,60)
seagreen (2e,8b,57)
seashell (ff,f5,ee)
sienna (a0,52,2d)
silver (c0,c0,c0)
skyblue (87,ce,eb)
slateblue (6a,5a,cd)
slategray (70,80,90)
snow (ff,fa,fa)
springgreen (00,ff,7f)
steelblue (46,82,b4)
tan (d2,b4,8c)
teal (00,80,80)
thistle (d8,bf,d8)
tomatao (ff,63,47)
turquoise (40,e0,d0)
violet (ee,82,ee)
wheat (f5,de,b3)
white (ff,ff,ff)
whitesmoke (f5,f5,f5)
yellow (ff,ff,00)
yellowgreen (9a,cd,32)

References

[1] Ronald C. Arkin. Motor Schema Based Navigation for a Mobile Robot: An Approach to Programming by Behavior. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 264–271, Raleigh, NC, March 1987.

[2] Ronald C. Arkin, William M. Carter, and Douglas C. Mackenzie. Active Avoidance: Escape and Dodging Behaviors for Reactive Control. *International Journal of Pattern Recognition and Artificial Intelligence*, 5(1):175–192, 1993.

[3] Michael R. Benjamin. The Interval Programming Model for Multi-Objective Decision Making. Technical Report AIM-2004-021, Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA, September 2004.

[4] Michael R. Benjamin. MOOS-IvP Autonomy Tools Users Manual. Technical Report MIT-CSAIL-TR-2010-039, MIT Computer Science and Artificial Intelligence Lab, August 2010.

[5] Michael R. Benjamin. MOOS-IvP Autonomy Tools Users Manual Release 13.2. Technical Report www.moos-ivp.org/docs.html, MIT Computer Science and Artificial Intelligence Lab, February 2013.

[6] Michael R. Benjamin, Paul M. Newman, Henrik Schmidt, and John J. Leonard. Extending MOOS-IvP and Users Guide to the IvPBuild Toolbox. Technical Report MIT-CSAIL-TR-2009-037, MIT Computer Science and Artificial Intelligence Lab, August 2009.

[7] Andrew A. Bennet and John J. Leonard. A Behavior-Based Approach to Adaptive Feature Detection and Following with Autonomous Underwater Vehicles. *IEEE Journal of Oceanic Engineering*, 25(2):213–226, April 2000.

[8] Rodney A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, April 1986.

[9] Marc Carreras, J. Batlle, and Pere Ridao. Reactive Control of an AUV Using Motor Schemas. In *International Conference on Quality Control, Automation and Robotics*, Cluj Napoca, Rumania, May 2000.

[10] George B. Dantzig. Programming in a Linear Structure. Comptroller, United States Air Force, February 1948.

[11] Oussama Khatib. Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 500–505, St. Louis, MO, March 1985.

[12] Ratnesh Kumar and James A. Stover. A Behavior-Based Intelligent Control Architecture with Application to Coordination of Multiple Underwater Vehicles. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Cybernetics*, 30(6):767–784, November 2001.

[13] Paul Newman. `http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php`.

[14] Paul M. Newman. MOOS - A Mission Oriented Operating Suite. Technical Report OE2003-07, MIT Department of Ocean Engineering, 2003.

[15] Paolo Pirjanian. *Multiple Objective Action Selection and Behavior Fusion.* PhD thesis, Aalborg University, 1998.

[16] Jukka Riekki. *Reactive Task Execution of a Mobile Robot.* PhD thesis, Oulu University, 1999.

[17] Julio K. Rosenblatt. *DAMN: A Distributed Architecture for Mobile Navigation.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1997.

[18] Julio K. Rosenblatt, Stefan B. Williams, and Hugh Durrant-Whyte. Behavior-Based Control for Autonomous Underwater Exploration. *International Journal of Information Sciences*, 145(1-2):69–87, 2002.

[19] Stefan B. Williams, Paul Newman, Gamini Dissanayake, Julio K. Rosenblatt, and Hugh Durrant-Whyte. A decoupled, distributed `AUV` control architecture. In *Proceedings of 31st International Symposium on Robotics*, pages 246–251, Montreal, Canada, 2000.

Index

Index of MOOS Variables