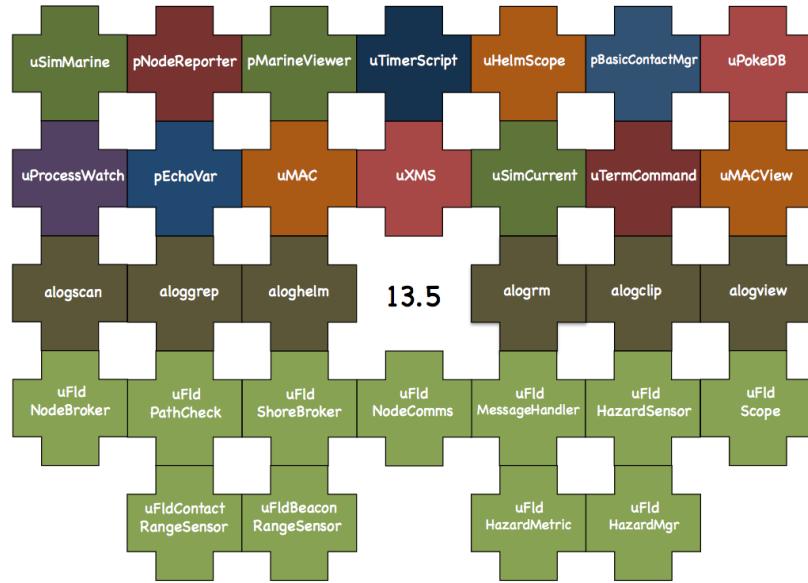


MOOS-IvP Autonomy Tools Users Manual

Release 13.5



Michael R. Benjamin
Department Mechanical Engineering
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge MA

May 01, 2013 - Release 13.5

Abstract

This document describes several MOOS-IvP autonomy tools. *uHelmScope* provides a run-time scoping window into the state of an active IvP Helm executing its mission. *pMarineViewer* is a geo-based GUI tool for rendering marine vehicles and geometric data in their operational area. *uXMS* is a terminal based tool for scoping on a MOOSDB process. *uTermCommand* is a terminal based tool for poking a MOOSDB with a set of MOOS file pre-defined variable-value pairs selectable with aliases from the command-line. *pEchoVar* provides a way of echoing a post to one MOOS variable with a new post having the same value to a different variable. *uProcessWatch* monitors the presence or absence of a set of MOOS processes and summarizes the collective status in a single MOOS variable. *uPokeDB* provides a way of poking the MOOSDB from the command line with one or more variable-value pairs without any pre-existing configuration of a MOOS file. *uTimerScript* will execute a pre-defined timed reusable script of poking variable-value pairs to a MOOSDB. *pNodeReporter* summarizes a platforms critical information into a single node report string for sharing beyond the vehicle. *pBasicContactMgr* provides a basic contact management service with the ability to generate range-dependent configurable alerts. *uSimMarine* provides

a simple marine vehicle simulator. The *Alog Toolbox* is a set of offline tools for analyzing and manipulating log files in the .alog format.



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:

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	Introduction	17
1.1	Overview of the MOOS-IvP Project and MOOS-IvP Tools	17
1.2	Module Overview	17
1.2.1	Mission Monitoring Modules	17
1.2.2	Mission Execution Modules	18
1.2.3	Mission Simulation Modules	18
1.2.4	Modules for Poking the MOOSDB	18
1.2.5	The Alog Toolbox	19
1.2.6	The uField Toolbox	19
1.2.7	Brief Background of MOOS-IvP	20
1.2.8	Sponsors of MOOS-IvP	21
1.2.9	The Software	21
1.2.10	Building and Running the Software	21
1.2.11	Operating Systems Supported by MOOS and IvP	23
1.2.12	Where to Get Further Information	23
1.3	The uField Toolbox	24
1.4	Motivations for the uField Toolbox	24
1.5	Introduction to AppCasting	26
1.5.1	Motivation For AppCasting	27
1.5.2	MOOS Applications and Terminal Output	27
1.5.3	Viewing AppCasts and Navigating AppCast Collections	28
1.5.4	The AppCast Data Structure	29
1.5.5	A Preview of AppCast Viewing Utilities	31
2	pMarineViewer: A GUI for Mission Monitoring and Control	33
2.1	Overview	33
2.1.1	The Shoreside-Vehicle Topology	33
2.1.2	Description of the pMarineViewer GUI Interface	35
2.1.3	The AppCasting, FullScreen and Traditional Display Modes	36
2.1.4	Run-Time and Mission Configuration	37
2.1.5	Command-and-Control	38
2.2	The BackView Pull-Down Menu	39
2.2.1	Panning and Zooming	39
2.2.2	Background Images	40
2.2.3	Local Grid Hash Marks	42
2.2.4	Full-Screen Mode	42
2.3	The GeoAttributes Pull-Down Menu	42
2.3.1	Polygons, SegLists, Points, Circles and Vectors	43
2.3.2	Markers	45
2.3.3	Comms Pulses	46
2.3.4	Range Pulses	48
2.3.5	Drop Points	49
2.4	The Vehicles Pull-Down Menu	50

2.4.1	The Vehicle Name Mode	50
2.4.2	Dealing with Stale Vehicles	51
2.4.3	Supported Vehicle Shapes	51
2.4.4	Vehicle Colors	52
2.4.5	Centering the Image According to Vehicle Positions	52
2.4.6	Vehicle Trails	52
2.5	The AppCast Pull-Down Menu	53
2.5.1	Turning On and Off AppCast Viewing	53
2.5.2	Adjusting the AppCast Viewing Panes Height and Width	54
2.5.3	Adjusting the AppCast Refresh Mode	54
2.5.4	Adjusting the AppCast Fonts	55
2.5.5	Adjusting the AppCast Color Scheme	55
2.6	The MOOS-Scope Pull-Down Menu	56
2.7	The Action Pull-Down Menu	56
2.8	The Mouse-Context Pull-Down Menu	58
2.8.1	Generic Poking of the MOOSDB with the Operation Area Position	58
2.8.2	Custom Poking of the MOOSDB with the Operation Area Position	58
2.9	The Reference-Point Pull-Down Menu	59
2.10	Configuration Parameters for pMarineViewer	61
2.10.1	Configuration Parameters for the BackView Menu	61
2.10.2	Configuration Parameters for the GeoAttributes Menu	62
2.10.3	Configuration Parameters for the Vehicles Menu	64
2.10.4	Configuration Parameters for the AppCast Menu	65
2.10.5	Configuration Parameters for the Scope, MouseContext and Action Menus	66
2.11	Publications and Subscriptions for pMarineViewer	66
2.11.1	Variables Published by pMarineViewer	66
2.11.2	Variables Subscribed for by pMarineViewer	67
3	uHelmScope: Scoping on the IvP Helm	68
3.1	Overview	68
3.2	The Helm Summary Section of the uHelmScope Output	69
3.2.1	The Helm Status (Lines 1-8)	69
3.2.2	The Helm Decision (Lines 9-11)	69
3.2.3	The Helm Behavior Summary (Lines 12-17)	69
3.3	The MOOSDB-Scope Section of the uHelmScope Output	70
3.4	The Behavior-Posts Section of the uHelmScope Output	71
3.5	Console Key Mapping and Command Line Usage	71
3.6	Helm-Produced Variables Used by uHelmScope	73
3.7	Configuration Parameters for uHelmScope	74
3.8	Publications and Subscriptions for uHelmScope	75
3.8.1	Variables Published by uHelmScope	75
3.8.2	Variables Subscribed for by uHelmScope	75
4	pNodeReporter: Summarizing a Node's Position and Status	77
4.1	Overview	77

4	Using pNodeReporter	78
4.2.1	Overview Node Report Components	78
4.2.2	Helm Characteristics	78
4.2.3	Platform Characteristics	79
4.2.4	Dealing with Local versus Global Coordinates	79
4.2.5	Processing Alternate Navigation Solutions	80
4.3	The Optional Blackout Interval Option	80
4.4	Configuration Parameters for pNodeReporter	82
4.5	Publications and Subscriptions for pNodeReporter	83
4.5.1	Variables Published by pNodeReporter	84
4.5.2	Variables Subscribed for by pNodeReporter	84
4.5.3	Command Line Usage of pNodeReporter	84
4.6	The Optional Platform Report Feature	85
4.7	An Example Platform Report Configuration Block for pNodeReporter	86
5	uXMS: Scoping the MOOSDB from the Console	88
5.1	Overview	88
5.2	The uXMS Refresh Modes	89
5.2.1	The Streaming Refresh Mode	89
5.2.2	The Events Refresh Mode	89
5.2.3	The Paused Refresh Mode	90
5.3	The uXMS Content Modes	90
5.3.1	The Scoping Content Mode	90
5.3.2	The History Content Mode	91
5.3.3	The Processes Content Mode	92
5.4	Configuration File Parameters for uXMS	94
5.4.1	The colormap Configuration Parameter	96
5.4.2	The content_mode Configuration Parameter	96
5.4.3	The display* Configuration Parameters	96
5.4.4	The history_var Configuration Parameter	97
5.4.5	The refresh_mode Configuration Parameter	98
5.4.6	The source Configuration Parameter	98
5.4.7	The term_report_interval Configuration Parameter	98
5.4.8	The trunc_data Configuration Parameter	98
5.4.9	The var Configuration Parameter	99
5.5	Command Line Usage of uXMS	99
5.6	Console Interaction with uXMS at Run Time	101
5.7	Running uXMS Locally or Remotely	101
5.8	Connecting multiple uXMS processes to a single MOOSDB	102
5.9	Using uXMS with Appcasting	102
5.10	Publications and Subscriptions for uXMS	103
5.10.1	Variables Published by uXMS	104
5.10.2	Variables Subscribed for by uXMS	104
6	uTimerScript: Scripting Events to the MOOSDB	105

6.1	Overview	105
6.2	Using uTimerScript	105
6.2.1	Configuring the Event List	106
6.2.2	Setting the Event Time or Range of Event Times	106
6.2.3	Resetting the Script	106
6.3	Script Flow Control	107
6.3.1	Pausing the Timer Script	107
6.3.2	Conditional Pausing of the Timer Script and Atomic Scripts	108
6.3.3	Fast-Forwarding the Timer Script	108
6.3.4	Quitting the Timer Script	108
6.4	Macro Usage in Event Postings	109
6.4.1	Built-In Macros Available	109
6.4.2	User Configured Macros with Random Variables	109
6.4.3	Support for Simple Arithmetic Expressions with Macros	110
6.5	Time Warps, Random Time Warps, and Restart Delays	110
6.5.1	Random Time Warping	110
6.5.2	Random Initial Start and Reset Delays	111
6.5.3	Status Messages Posted to the MOOSDB by uTimerScript	111
6.6	Terminal and AppCast Output	112
6.7	Configuration File Parameters for uTimerScript	113
6.8	Publications and Subscriptions for uTimerScript	114
6.8.1	Variables Published by uTimerScript	114
6.8.2	Variables Subscribed for by uTimerScript	115
6.8.3	An Example MOOS Configuration Block	115
6.9	Examples	116
6.9.1	A Script Used as Proxy for an On-Board GPS Unit	116
6.9.2	A Script as a Proxy for Simulating Random Wind Gusts	118
7	pBasicContactMgr: Managing Platform Contacts	121
7.1	Overview	121
7.2	Using pBasicContactMgr	121
7.2.1	Contact Alert Configuration	122
7.2.2	Dynamic Contact Alert Configuration	123
7.2.3	Contact Alert Triggers	124
7.2.4	Contact Alert Record Keeping	125
7.2.5	Contact Resolution	126
7.3	Deferring to Earth Coordinates over Local Coordinates	126
7.4	Usage of the pBasicContactMgr with the IvP Helm	127
7.5	Terminal and AppCast Output	127
7.6	Configuration Parameters for pBasicContactMgr	128
7.6.1	An Example MOOS Configuration Block	129
7.7	Publications and Subscriptions for pBasicContactMgr	130
7.7.1	Variables Published by pBasicContactMgr	130
7.7.2	Variables Subscribed for by pBasicContactMgr	131
7.7.3	Command Line Usage of pBasicContactMgr	131

8	uProcessWatch: Monitoring MOOS Application Health	133
8.1	Overview	133
8.2	Typical uProcessWatch Usage Scenarios	134
8.2.1	Using uProcessWatch with AppCasting and pMarineViewer	134
8.2.2	Directly Accessing the PROC_WATCH_SUMMARY Output	134
8.3	Using and Configuring the uProcessWatch Utility	135
8.3.1	The DB_CLIENTS Variable for Detecting Missing Processes	135
8.3.2	Defining the Watch List	135
8.3.3	Reports Generated	136
8.3.4	Watching and Reporting on a Single MOOS Process	136
8.3.5	A Heartbeat for the Watch Dog	137
8.3.6	Excusing a Process	137
8.3.7	Allowing Retractions if a Process Reappears	138
8.4	Configuration Parameters of uProcessWatch	139
8.4.1	An Example MOOS Configuration Block	139
8.5	Publications and Subscriptions for uProcessWatch	140
8.5.1	Variables Published by uProcessWatch	140
8.5.2	MOOS Variables Subscribed for by uProcessWatch	140
9	uSimMarine: Basic Vehicle Simulation	142
9.1	Configuration Parameters for uSimMarine	142
9.2	Publications and Subscriptions for uSimMarine	144
9.2.1	Variables Published by uSimMarine	144
9.2.2	Variables Subscribed for by uSimMarine	145
9.2.3	Command Line Usage of uSimMarine	145
9.3	Setting the Initial Vehicle Position, Pose and Trajectory	146
9.4	Propagating the Vehicle Speed, Heading, Position and Depth	146
9.4.1	Propagating the Vehicle Speed	147
9.4.2	Propagating the Vehicle Heading	148
9.4.3	Propagating the Vehicle Position	149
9.4.4	Propagating the Vehicle Depth	150
9.5	Propagating the Vehicle Altitude	151
9.6	Simulation of External Drift	152
9.6.1	External X-Y Drift from Initial Simulator Configuration	152
9.6.2	External X-Y Drift Received from Other MOOS Applications	152
9.7	The ThrustMap Data Structure	154
9.7.1	Automatic Pruning of Invalid Configuration Pairs	154
9.7.2	Automatic Inclusion of Implied Configuration Pairs	155
9.7.3	A Shortcut for Specifying the Negative Thrust Mapping	155
9.7.4	The Inverse Mapping - From Speed To Thrust	155
9.7.5	Default Behavior of an Empty or Unspecified ThrustMap	156
10	The uMAC Utilities	157
10.1	The uMACView Utility	157
10.1.1	Publications and Subscriptions	158

10.1.2	Configuration File Parameters	158
10.1.3	Command Line Arguments and Options	159
10.1.4	Refresh Modes	159
10.2	The uMAC Utility	160
10.2.1	Content Modes	160
10.2.2	Refresh Modes	162
10.2.3	A Tip Regarding Process Monitoring and uMAC Sessions	162
10.2.4	Publications and Subscriptions	162
10.2.5	Configuration File Parameters	162
10.2.6	Command Line Arguments and Options	162
10.3	The uMACView Utility Integrated with pMarineViewer	163
11	Enabling a MOOS Application for AppCasting	164
11.1	Sub-classing the AppCastingMOOSApp Superclass	164
11.2	Invoking Superclass Methods in the Iterate() Method	165
11.3	Invoking a Superclass Method in the OnNewMail() Method	165
11.4	Invoking a Superclass Method in the OnStartUp() Method	166
11.5	Invoking a Superclass Method When Registering for Variables	166
11.6	Implementing a buildReport Method for Generating AppCasts	166
11.7	Posting Events	168
11.8	Posting Run Warnings	168
11.9	Posting Configuration Warnings	169
11.10	Under The Hood - On-Demand AppCasting	172
11.10.1	Motivation	172
11.10.2	AppCast Generation Criteria	172
11.10.3	Terminal Switching	173
11.10.4	AppCast Requests	173
11.10.5	Limiting the AppCast Frequency	175
11.10.6	Generating and AppCast vs. Publishing and AppCast	175
11.10.7	Monitoring AppCast Traffic Volume	176
12	pHostInfo: Detecting and Sharing Host Info	177
12.1	Configuration Parameters for pHostInfo	178
12.2	Publications and Subscriptions for pHostInfo	178
12.2.1	Variables Published by pHostInfo	178
12.2.2	Variables Subscribed for by pHostInfo	179
12.2.3	Command Line Usage of pHostInfo	179
12.3	Usage Scenarios the pHostInfo Utility	180
12.3.1	Handling Multiple IP Addresses	180
12.4	A Peek Under the Hood	180
12.4.1	Temporary Files	180
12.4.2	Possible Gotchas	180
13	uPokeDB: Poking the MOOSDB from the Command Line	182
13.1	Overview	182

13.2	Command-line Arguments of uPokeDB	182
13.3	MOOS Poke Macro Expansion	183
13.4	Providing the ServerHost and ServerPort on the Command Line	183
13.5	Session Output from uPokeDB	184
13.6	Publications and Subscriptions for uPokeDB	184
14	pEchoVar: Re-publishing Variables Under a Different Name	186
14.1	Overview	186
14.2	Using pEchoVar	186
14.2.1	Configuring Echo Mapping Events	186
14.2.2	Configuring Flip Mapping Events	186
14.2.3	Applying Conditions to the Echo and Flip Operation	188
14.2.4	Holding Outgoing Messages Until Conditions are Met	188
14.2.5	Limiting the Echo Posting Frequency to the AppTick Setting	189
14.3	Configuring for Vehicle Simulation with pEchoVar	189
14.4	Configuration Parameters for pEchoVar	189
14.5	Publications and Subscriptions for pEchoVar	190
14.5.1	Variables Posted by pEchoVar	190
14.5.2	Variables Subscribed for by pEchoVar	190
14.6	Terminal and AppCast Output	190
15	pSearchGrid: Using a 2D Grid Model for Track History	193
15.1	Using pSearchGrid	193
15.1.1	Basic Configuration of Grid Cells	194
15.1.2	Cell Variables	194
15.1.3	Serializing and De-serializing the Grid Structure	194
15.1.4	Resetting the Grid	195
15.1.5	Viewing Grids in pMarineViewer	195
15.1.6	Examples	195
15.2	Configuration Parameters of pSearchGrid	195
15.3	Publications and Subscriptions for pSearchGrid	196
15.3.1	Variables Published by pSearchGrid	196
15.3.2	Variables Subscribed for by pSearchGrid	197
15.3.3	Command Line Usage of pSearchGrid	197
16	uTermCommand: Poking the MOOSDB with Pre-Set Values	198
16.1	Configuration Parameters for uTermCommand	198
16.2	Run Time Console Interaction	199
16.3	Connecting uTermCommand to the MOOSDB Under an Alias	200
16.4	Publications and Subscriptions for uTermCommand	200
17	uSimCurrent: Simulating Drift Effects	201
17.1	Configuration Parameters for uSimCurrent	201
17.2	Publications and Subscriptions for uSimCurrent	201
17.2.1	MOOS Variables Published by uSimCurrent	202
17.2.2	MOOS Variables Subscribed for by uSimCurrent	202

18 The Alog-Toolbox for Analyzing and Editing Mission Log Files	203
18.1 Overview	203
18.2 An Example .alog File	203
18.3 The <code>alogscan</code> Tool	203
18.3.1 Command Line Usage for the <code>alogscan</code> Tool	203
18.3.2 Example Output from the <code>alogscan</code> Tool	204
18.4 The <code>alogclip</code> Tool	206
18.4.1 Command Line Usage for the <code>alogclip</code> Tool	206
18.4.2 Example Output from the <code>alogclip</code> Tool	207
18.5 The <code>aloggrep</code> Tool	207
18.5.1 Command Line Usage for the <code>aloggrep</code> Tool	208
18.5.2 Example Output from the <code>aloggrep</code> Tool	208
18.6 The <code>alogrm</code> Tool	209
18.6.1 Command Line Usage for the <code>alogrm</code> Tool	209
18.6.2 Example Output from the <code>alogrm</code> Tool	210
18.7 The <code>alogview</code> Tool	211
18.7.1 Command Line Usage for the <code>alogview</code> Tool	212
18.7.2 Description of Panels in the <code>alogview</code> Window	213
18.7.3 The Op-Area Panel for Rendering Vehicle Trajectories	213
18.7.4 The Helm Scope Panels for View Helm State by Iteration	215
18.7.5 The Data Plot Panel for Logged Data over Time	216
18.7.6 Automatic Replay of the Log file(s)	216
19 uFldNodeBroker: Brokering Node Connections	217
19.1 The uFldNodeBroker Interface and Configuration Options	218
19.1.1 Configuration Parameters of uFldNodeBroker	218
19.2 Publications and Subscriptions for uFldNodeBroker	219
19.2.1 Variables Published by uFldNodeBroker	219
19.2.2 Variables Subscribed for by uFldNodeBroker	219
19.2.3 Command Line Usage of uFldNodeBroker	220
19.3 Terminal and AppCast Output	220
20 uFldShoreBroker: Brokering Shore Connections	223
20.1 Bridging Variables Upon Connection to Nodes	224
20.1.1 Inter-MOOSDB Bridging with pShare	224
20.1.2 Handling a Valid Incoming Ping from a Remote Node	224
20.1.3 Vanilla Bridge Arrangements	225
20.1.4 Bridge Arrangements with Macros	226
20.1.5 A Common Configuration Shortcut - the qbridge Parameter	226
20.2 Usage Scenarios for the uFldShoreBroker Utility	227
20.3 Terminal and AppCast Output	227
20.4 Configuration Parameters of uFldShoreBroker	229
20.5 Publications and Subscriptions for uFldShoreBroker	230
20.5.1 Variables Published by uFldShoreBroker	230
20.5.2 MOOS Variables Subscribed for by uFldShoreBroker	230

20.5.3 Command Line Usage of uFldShoreBroker	230
21 uFldNodeComms: Simulating Intervehicle Communications	232
21.1 Handling Node Reports	233
21.1.1 The Criteria for Routing Node Reports	233
21.1.2 Node Report Transmissions and pShare	235
21.2 Handling Node Messages	235
21.2.1 The Criteria for Routing Node Messages	235
21.2.2 Enforcing a Minimum Time Between Node Messages	236
21.2.3 Enforcing a Maximum Node Message Length	236
21.2.4 Posting Messages to a Vehicle Group	236
21.3 Visual Artifacts for Rendering Inter-Vehicle Communications	236
21.4 Terminal and AppCast Output	237
21.5 Configuration Parameters of uFldNodeComms	238
21.6 Publications and Subscriptions for uFldNodeComms	240
21.6.1 Variables Published by uFldNodeComms	240
21.6.2 Variables Subscribed for by uFldNodeComms	240
22 uFldMessageHandler: Handling Incoming Node Messages	242
22.1 Configuration Parameters of uFldMessageHandler	243
22.2 Publications and Subscriptions for uFldMessageHandler	243
22.2.1 Variables Published by uFldMessageHandler	244
22.2.2 Variables Subscribed for by uFldMessageHandler	244
22.3 Terminal and AppCast Output	245
23 uFldScope: Gathering a Multi-Vehicle Status Summary	247
23.1 Configuration Parameters of uFldScope	248
23.1.1 An Example MOOS Configuration Block	248
23.2 Publications and Subscriptions for uFldScope	248
23.2.1 Variables Published by uFldScope	249
23.2.2 MOOS Variables Subscribed for by uFldScope	249
23.3 Configuring the uFldScope Utility	249
23.3.1 Configuring Scope Elements	250
23.3.2 Configuring Scope Layouts	250
23.3.3 Further Control of the Terminal Output	251
24 uFldPathCheck: Monitoring Vehicle Path Properties	252
24.1 Overview of the uFldPathCheck Interface and Configuration Options	252
24.1.1 Configuration Parameters of uFldPathCheck	253
24.2 Publications and Subscriptions for uFldPathCheck	253
24.2.1 Variables Published by uFldPathCheck	253
24.2.2 Variables Subscribed for by uFldPathCheck	253
24.2.3 An Example MOOS Configuration Block	254
24.3 Usage Scenarios the uFldPathCheck Utility	254
25 uFldHazardSensor: Simulating an Simple Hazard Sensor	255

25.1	A Quick Start Guide to Using uFldHazardSensor	255
25.1.1	A Working Example Mission - the Jake Mission	256
25.1.2	A Bare-Bones Example uFldHazardSensor Configuration	257
25.1.3	A Simple Hazard File	257
25.1.4	Typical Simulator Topology	258
25.2	Detections and Classifications	260
25.2.1	The Simulated Detection Algorithm	260
25.2.2	The Simulated Classification Algorithm	261
25.3	Simulator Sensor Configuration Options	263
25.3.1	Sensor Swath Width Options	263
25.3.2	Sensor ROC Curve Configuration Options	264
25.3.3	Classification Configuration Options	265
25.3.4	Dynamic Resetting of the Sensor	266
25.3.5	Posting of Sensor Configuration Options	266
25.4	Configuring the Hazard Field	267
25.4.1	An Example Hazard Field	268
25.4.2	Automatically Generating a Hazard Field	269
25.5	Hazard Resemblance Factor	269
25.5.1	The Effect of the Resemblance Factor on Detections	269
25.5.2	The Effect of the Resemblance Factor on Classifications	269
25.5.3	Generating a Random Hazard Field with Resemblance Factors	270
25.6	Aspect Angle Sensitivity	270
25.6.1	The Effect of Aspect Sensitivity on the Detection Algorithm	271
25.6.2	The Effect of Aspect Sensitivity on the Classification Algorithm	272
25.6.3	Visual Cues on Aspect Sensitivity	272
25.7	Configuring the Simulator Visual Preferences	272
25.7.1	Configuring the Sensor Field Swath Rendering	272
25.7.2	Configuring the Hazard Field Renderings	273
25.7.3	Configuring the Sensor Report Renderings	273
25.7.4	Rendering the Prevailing P_D and P_{FA} Values	273
25.8	Under the Hood: Sensor Blackouts During Turns	274
25.9	Configuration Parameters of uFldHazardSensor	275
25.9.1	An Example MOOS Configuration Block	276
25.10	Publications and Subscriptions for uFldHazardSensor	277
25.10.1	Variables Published by uFldHazardSensor	277
25.10.2	Variables Subscribed for by uFldHazardSensor	278
25.11	Terminal and AppCast Output	279
25.12	The Jake Example Mission Using uFldHazardSensor	280
25.12.1	What is Happening in the Jake Mission	280
26	uFldHazardMgr: On-Board Management of a Hazard Sensor	283
26.1	Overview	283
26.2	Using uFldHazardMgr	284
26.2.1	Required MOOS Variable Bridges	285
26.2.2	Configuration Parameters of uFldHazardMgr	285

26.2.3	An Example MOOS Configuration Block	286
26.2.4	Configuring the Swath Width	286
26.2.5	Configuring the Probability of Detection Setting	287
26.3	Under the Hood - Interacting with the Hazard Sensor	287
26.4	Under the Hood - Processing Data and Generating Reports	287
26.5	Publications and Subscriptions for uFldHazardMgr	288
26.5.1	Variables Published by uFldHazardMgr	288
26.5.2	Variables Subscribed for by uFldHazardMgr	288
26.6	Terminal and AppCast Output	289
26.7	The Jake Example Mission Using uFldHazardMgr	290
27	uFldHazardMetric: Grading a HazardSet Report	291
27.1	Overview	291
27.2	Using uFldHazardMetric	291
27.2.1	Required MOOS Variable Bridges	292
27.2.2	The False-Alarm and Missed-Hazard Reward Structure	293
27.2.3	The Max-Time and Time-Overage Reward Structure	293
27.2.4	Raw and Normalized Scores	294
27.2.5	The Report Evaluation Format	294
27.3	Configuration Parameters of uFldHazardMetric	295
27.4	Publications and Subscriptions for uFldHazardMetric	296
27.4.1	Variables Published by uFldHazardMetric	296
27.4.2	Variables Subscribed for by uFldHazardMetric	296
27.5	Terminal and AppCast Output	297
27.6	The Jake Example Mission Using uFldHazardMetric	299
28	The Jake Example Mission	300
28.1	Module Topology in the Jake Mission	301
28.2	Key MOOS Variables in the Jake Mission	302
28.3	General Description of Events in the Jake Mission	303
28.3.1	Phase 1: Sensor Configuration and Handling of Mission Parameters	303
28.3.2	Phase 2: Executing the Search Phase	304
28.3.3	Phase 3: Reporting Results	305
28.4	Required MOOS Variable Bridges	305
28.4.1	Variable Bridges from the Shoreside to the Vehicle	306
28.4.2	Variable Bridges from the Vehicle to the Shoreside	306
28.5	Hazard Files in the Jake Mission	307
29	uFldBeaconRangeSensor: Simulating Vehicle to Beacon Ranges	308
29.1	The uFldBeaconRangeSensor Interface and Configuration Options	309
29.1.1	Configuration Parameters of uFldBeaconRangeSensor	309
29.2	Publications and Subscriptions for uFldBeaconRangeSensor	311
29.2.1	Variables Published by uFldBeaconRangeSensor	311
29.2.2	Variables Subscribed for by uFldBeaconRangeSensor	312
29.3	Using and Configuring uFldBeaconRangeSensor	312

29.3.1	Configuring the Beacon Locations and Properties	314
29.3.2	Unsolicited Beacon Range Reports	315
29.3.3	Solicited Beacon Range Reports	316
29.3.4	Limiting the Frequency of Vehicle Range Requests	316
29.3.5	Producing Range Measurements with Noise	317
29.3.6	Terminal and AppCast Output	317
29.4	Interaction between uFldBeaconRangeSensor and pMarineViewer	319
29.4.1	The VIEW_MARKER Data Structure	319
29.4.2	The VIEW_RANGE_PULSE Data Structure	320
29.5	The Indigo Example Mission Using uFldBeaconRangeSensor	320
29.5.1	Examining the Log Data from the Indigo Mission	321
29.5.2	Generating Range Report Data for Matlab	322
30	uFldContactRangeSensor: Detecting Contact Ranges	323
30.1	Overview	323
30.2	Using uFldContactRangeSensor	323
30.2.1	Typical Topology	323
30.2.2	Required MOOS Variable Bridges	324
30.2.3	Range Requests and Range Reports	325
30.2.4	Configuring the Range Criteria	326
30.2.5	Adding Exponentially Decaying Detection Probability Beyond Range	326
30.2.6	Limiting the Arcs of Vehicle Range Requests	327
30.2.7	Limiting the Frequency of Vehicle Range Requests	328
30.2.8	Producing Range Measurements with Noise	328
30.3	Configuration Parameters of uFldContactRangeSensor	329
30.4	Publications and Subscriptions for uFldContactRangeSensor	330
30.4.1	Variables Published by uFldContactRangeSensor	331
30.4.2	Variables Subscribed for by uFldContactRangeSensor	331
30.5	Terminal and AppCast Output	332
30.6	Interaction between uFldContactRangeSensor and pMarineViewer	333
30.7	The Hugo Example Mission Using uFldContactRangeSensor	333
31	uFldCollisionDetect: Detecting Collisions	336
31.1	Overview	336
31.2	Using uFldCollisionDetect	336
31.2.1	Configuring the Collision Distance	336
31.2.2	AppCasting with uFldCollisionDetect	336
31.2.3	Range Pulses with uFldCollisionDetect	337
31.3	Configuration Parameters of uFldCollisionDetect	338
31.4	Publications and Subscriptions for uFldCollisionDetect	340
31.4.1	Variables Published by uFldCollisionDetect	340
31.4.2	Vehicle-Specific Publication	340
31.4.3	Generic Collision Publication	340
31.4.4	Variables Subscribed for by uFldCollisionDetect	341

A Use of Logic Expressions	342
B Colors	344

1 Introduction

1.1 Overview of the MOOS-IvP Project and MOOS-IvP Tools

The MOOS-IvP autonomy tools described in this document are software applications that are typically running either as part of an overall autonomy system running on a marine vehicle, as part of a marine vehicle simulation, or used for post-mission off-line analysis. They are each MOOS applications, meaning they are running and communicating with a MOOSDB. The AlogToolbox described here contains a number of off-line tools for analyzing alog files produced by the pLogger application.

The focus of this paper is on these tools, and the set of off-line mission analysis tools comprising the Alog Toolbox. Important topics outside this scope are (a) MOOS middleware programming, (b) the IvP Helm and autonomy behaviors, and (c) other important MOOS utilities applications not covered here. The intention of this paper is to provide documentation for these common applications for current users of the MOOS-IvP software.

1.2 Module Overview

1.2.1 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 2.
- *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 3.
- *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 processes on which all variables published by those processes will be scoped. Users may also scope on the *history* of a single variable. Section 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. Section 8.
- *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. Section 10.2.

- *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. Section 10.1.

1.2.2 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. Section 4.
- *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 7.
- *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. Section 14.
- *pSearchGrid*: An application for storing a history of vehicle positions in a 2D grid defined over a region of operation. Section 15.

1.2.3 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. Section 9.
- *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**. Section 17.

1.2.4 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. Section 13.
- *uTimerScript*: Allows the user to script a set of pre-configured pokes to a MOOSDB with each entry in the script happening after a specified 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 6.
- *uTermCommand*: A terminal application for poking the MOOSDB with pre-defined variable-value pairs. A unique key may be associated with each poke. Section 16.

1.2.5 The Alog Toolbox

The Alog Toolbox is set of offline tools for analyzing and manipulating alog files produced 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. Section 18.3.
- *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. Section 18.4.
- *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. Section 18.5.
- *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. Section 18.6.
- *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). Section 18.7.

1.2.6 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. Section 12.
- *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. Section 19.



- *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. Section 20.
- *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. Section 21.
- *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. Section 22.
- *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. Section 23.
- *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. Section 24.
- *uFldHazardSensor*: Typically run in a shoreside community. Configured with a set objects with a given x,y location and classification (hazard or benign). The sensor simulator receives a series of requests from a remote vehicle. When sensor determines that an object 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. Section 25.
- *uFldHazardMetric*: An application for grading incoming hazard reports, presumably generated by users of the uFldHazardSensor after exploring a simulated hazard field. Section 27.
- *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. Section 26.
- *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. Section 29.
- *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. Section 30.

1.2.7 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.

1.2.8 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. The ILIR program is funded by ONR.

1.2.9 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.

1.2.10 Building and Running the Software

This document is written to Release 13.5. 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_V10Beta_rc2    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 `10Beta_rc2`. 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*: Ubiquitous 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 middleware 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.

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

Alternatively one can go directly into the `MOOS` directory and configure options with `cmake` 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 directory containing the MOOS executables:

`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.

1.2.11 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.

1.2.12 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:

<http://www.themoos.org>

<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:



1.3 The uField Toolbox

The uField Toolbox is a set of tools to facilitate the deployment of and simulation of multiple fielded marine vehicles. It pre-supposes the arrangement depicted in Figure 1 below. A number of vehicles are deployed and are connected to a single shoreside command and control computer. Each vehicle, as well as the shoreside computer, contain a dedicated MOOS community. A community is comprised of a MOOSDB process and a number of connected MOOS applications.

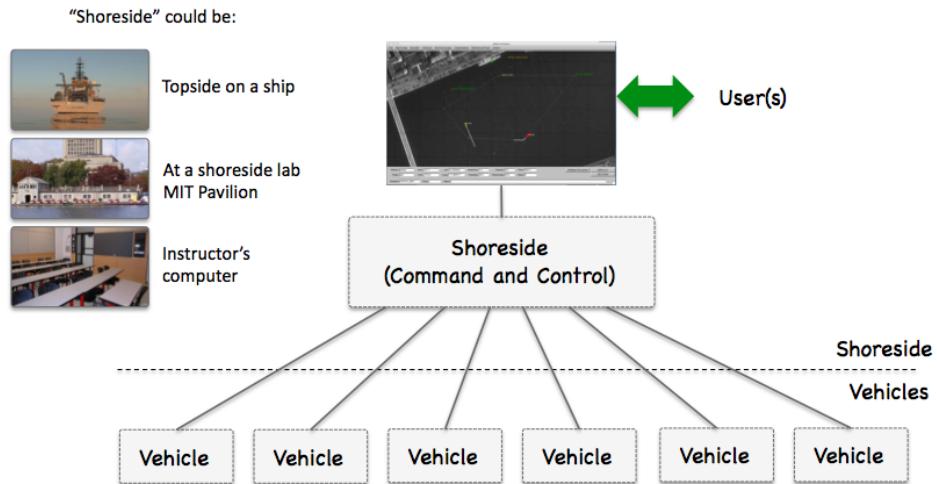


Figure 1: **Shoreside to Multi-Vehicle Topology:** A number of vehicles are deployed with each vehicle maintaining some level of connectivity to a shoreside command and control computer. Each node (vehicles and the shoreside) are comprised of a dedicated MOOS community. Modes and limits of communication may vary.

The uField Toolbox is designed to support both laboratory simulations as well as fielded experiments with multiple marine vehicles.

The uField Toolbox in Laboratory Simulations

In laboratory or classroom simulations, the nodes in Figure 1 may be either (a) all running on a single user's machine during initial development and testing, or (b) each running on a separate laptop with the shoreside community rendered on a central laboratory screen.

The uField Toolbox in Fielded Experimentation

During field experiments, the vehicle nodes in Figure 1 are actual fielded vehicles. The shoreside node is a machine on the shore with connectivity to each of the vehicles. For example, at the MIT Sailing Pavilion lab, the vehicles are either kayaks, kingfishers or a man-portable UUV. Communications is typically achieved with WiFi, and occasionally over acoustic link, or cellular network link.

1.4 Motivations for the uField Toolbox

The uField Toolbox is meant to facilitate and simulate. It facilitates aspects of configuration for inter-vehicle communications with [pShare](#) by letting nodes auto-configure with one another. It

facilitates the monitoring of properties of several fielded or simulated vehicles by collecting and displaying user configurable information. It simulates properties inter-vehicle communication such as range dependency and bandwidth limitations. It simulates inter-vehicle contact detection as would otherwise occur through AIS or vehicle mounted sensors. And it simulates a range-only sensor to fixed beacon locations and moving contacts.

Facilitation of Inter-Vehicle Connections

 When vehicles are on the internet, or common local subnet, their inter-vehicle communications are handled by `pShare`. This is the case during multi-vehicle simulations on a single machine, multi-vehicle simulations between a room full of laptops, and field exercises at a facility like the MIT Sailing Pavilion when all vehicles are connected by a local WiFi router. It is even the case when vehicles are connected using the cellular telephone network. When or if a vehicle is connected via an acoustic modem, or via a satellite link, `pShare` is replaced with a dedicated device interface.

A hurdle to using `pShare` has been the configuration of `pShare` on each machine to properly point the IP address and community name of the vehicles to which it wishes to communicate. If the IP address and vehicle/community names of all machines never change, this may be tractible but cumbersome. When the IP addresses are not known at run time, the configuration problem is often a deal-breaker. In conjunction with the development of the uField Toolbox, a key modification to `pShare` was made to allow it to accept incoming mail specifying new bridge configurations. The `uFldNodeBroker` app may be run on each vehicle to communicate in a generic manner to the shoreside MOOS community running `uFldShoreBroker`. These two apps communicate to share IP and host information and request modifications to their own local `pShare` instances, automatically configuring them for all further communications. All that is needed on the vehicles is a list of possible shoreside IP addresses to try for initial connections. Collectively these tools greatly simplify the `pShare` comms configurations in both simulation and fielded exercises.

Facilitation of Multi-Vehicle Status Fields

Deploying multiple vehicles makes it increasingly important to have access to key information across all vehicles in a concise, user-configurable format. For example, it may be good to know which vehicles have had start-up problems, which vehicles are running low on power, or the total trip distance or maximum noted speed for all vehicles. The `uFldScope` tool is built to address this. It exploits a convention that many MOOS status messages are in the format of comma-separated parameter=value pairs. The `uFldScope` tool may be configured to register for any such status message, provided a field to scope, and provided a field indicating the vehicle name. It produces a series of terminal-output reports in a simple tabular format with a row for each vehicle and column for each scoped piece of information. The tool may be configured to produce reports on multiple different sets of scoped data, allowing the user to toggle through the reports.

Simulating Inter-Vehicle Contact Detection

Detecting the position of another vehicle may be necessary (a) for performing collision avoidance, (b) for coordination with a collaborating vehicle, or (c) for stategizing against a competitor vehicle. Knowing another vehicle's position may come from either communication of vehicle position

through a general system such as AIS, through direct vehicle to vehicle communication, or through on-board sensors uses to detect and track another vehicle.

The goal of the `uFldNodeComms` utility is simulate the above scenarios. It accepts node reports from *all* fielded or simulated vehicles, and passes on this information to other vehicles. The user may choose a configuration where every vehicle knows the position of every other vehicle all the time. Or a configuration may be chosen where vehicle report sharing is based on the vehicle range. The user may also choose a configuration where vehicle group or team membership determines, in part, the sharing of node reports.

Simulating Range-Only Sensors

A set of common sensing problems for marine vehicles include (a) determining the location of a fixed object given only a sequence of range-only measurements, (b) determining the location of one's own vehicle given range-only measurements to an object with a known location, and (c) determining the location and trajectory of a moving contact given a sequence of range-only measurements to the contact.

The uField Toolbox includes two tools for simulating these range-only sensors, the `uFldBeaconRangeSensor` and `uFldContactRangeSensor` applications. Each tool is situated on the shoreside computer and awaits the arrival of a “range request” from a deployed or simulated vehicle. The shoreside computer is also receiving node reports from all vehicles and is thus able to factor in relative vehicle position in determining whether or not to reply. Range replies contain the information otherwise perhaps derived from a range-only sensor residing on the vehicle.

Simulating Inter-Vehicle Message Transmission

Message transmission between vehicles is never a given. There is almost always some degree of range dependency, and some degree of uncertainty at any range as to whether the message will get through. The message length itself may also be limited depending on how it is sent. Underwater messaging is particularly subject to all three considerations.

The `uFldNodeComms` utility is designed to simulate message passing between vehicles. This utility resides on a shoreside computer and operates by requiring inter-vehicle message passing to be routed through this utility back out to the receiving vehicles. Since `uFldNodeComms` also is receiving node reports from all fielded vehicles, it may apply the relative position of vehicles in its determination of whether or not to send the message. It may also arbitrarily restrict the size of the message being sent.

1.5 Introduction to AppCasting

AppCasting provides an *optional* new way of delivering application status output beyond the traditional means of writing to standard I/O in a terminal window. It is motivated by a few common recurring observations:

- The biggest headache of new users to MOOS (e.g., students in MIT 2.680) was the derailment of a mission due to an unnoticed configuration or runtime error.
- Debugging typically involves re-launching with app terminal windows open and analyzing expected vs. observed output.

- When deploying multiple simulated vehicles, each with multiple MOOS apps, the number of open terminal windows may be unmanageable.
- When deploying a vehicle in the field, one cannot ssh in and see any application terminal output at all.
- Since terminal output is rarely viewable for the above practical reasons, apps are rarely designed with much thought put into their status output.

1.5.1 Motivation For AppCasting

AppCasting is designed to make it easier to see application terminal output. This includes app-specific status messages, configuration and runtime warnings, and notable events. It is designed to allow appcast viewing tools to render this information and alerts on a single screen across multiple vehicles, each with several running apps, whether in simulation or the field. The belief is that having this form of information easier at hand, application developers will find it more rewarding to add thoughtful status reports to their application's functionality. The applications in the MOOS-IvP tree have been almost all converted to support appcasting. It's worth repeating that this is an opt-in feature of MOOS. All existing MOOS apps work just fine without appcasting. Appcasting apps and non-appcasting apps work side-by-side seamlessly.

1.5.2 MOOS Applications and Terminal Output

A MOOS application typically interacts with the world primarily through its subscriptions and publications to the MOOSDB. It may also interact through a terminal interface by generating status or debugging output, or in some rare cases, accepting terminal input.

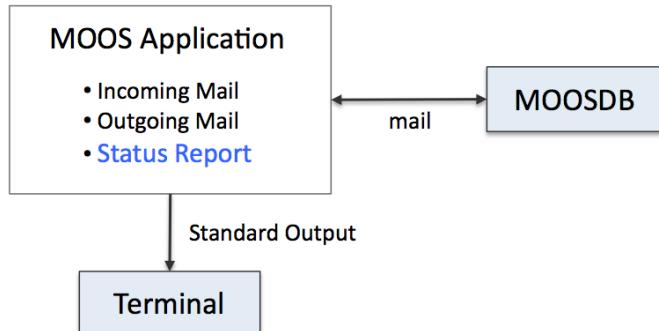


Figure 2: A typical MOOS application interacts with the world by publishing and receiving mail through the MOOSDB and by writing status messages to a terminal window, if one is open.

Even MOOS GUI applications have the option of opening a terminal interface in addition to the GUI. The terminal interface option is invoked by setting `NewConsole=true` for the given app in the `ANTLER` process configuration block in the mission file, or simply by just launching the app manually from the command line. With AppCasting, an application still generates terminal output, but does so by creating a report, in the form of an appcast data structure. The data structure may be converted to a list of strings, sent to the terminal, or a single long string, published to the MOOSDB.

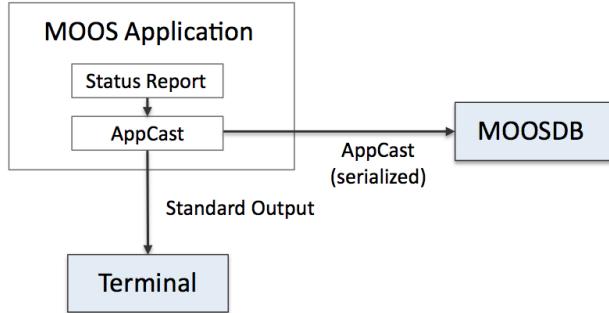


Figure 3: An appcasting MOOS app produces terminal by repeatedly generating and sending an *appcast* to the terminal. The appcast is also serialized and published to the MOOSDB for other MOOS applications to consume.

By sending the appcast to the MOOSDB this makes a few things possible. First, even if the application was launched long ago without a terminal, it is now possible to launch a separate MOOS application (an appcast viewer tool discussed in Section 1.5.5) to start looking at the status output. Second, the same appcast data structure may now also be bridged to a separate off-board MOOS community, using something like `pShare`, so remote users may be alerted to or debug problems. Third, the appcast data structure may contain configuration and run-time *alerts* to bring issues to the attention of operators quickly. Fourth, the appcast structure may be logged like any other MOOS variable, making it possible to review terminal output during the post-mission analysis phase.

1.5.3 Viewing AppCasts and Navigating AppCast Collections

A primary motivation for appcasting is the ability to view appcasts over several applications with a single viewer:

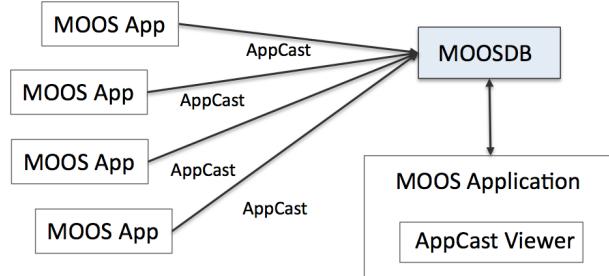


Figure 4: An AppCast Viewer is a separate MOOS application for gathering and navigating through appcasts from several other MOOS applications.

In addition to being able to see application output from a single window, it is possible to view application output for a particular app regardless of how that application was launched. In comparison, if the status output were only viewable from a terminal window, that app would have had to be launched with a terminal window from the outset. Furthermore, when the AppCast Viewer has a terminal interface, a user may log onto a remotely deployed vehicle and launch the viewer and see application output that would not be viewable otherwise since applications on fielded

platforms are never run with terminal windows open.

An AppCast Viewer may also handle appcasts from several vehicles as conveyed in Figure 5. The viewer lets the user navigate between different vehicles and appcasts within a vehicle. The interface is discussed in a later section, but the idea is shown on the right in Figure 7.

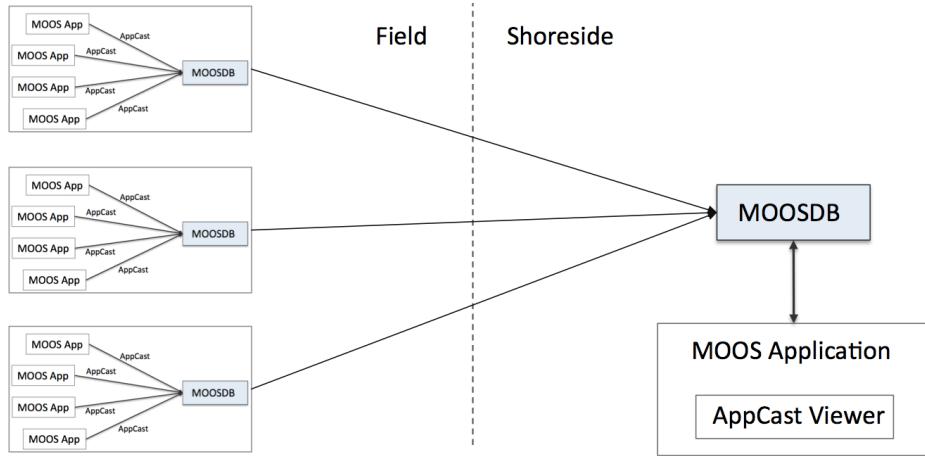


Figure 5: An AppCast Viewer may also be used for sorting and navigating through appcasts from several applications *over several vehicles or nodes*. The appcasts may be bridged from one community to a single shoreside community using a tool such as pShare.

The above arrangement assumes that appcasts are sent from each MOOS community to a single *shoreside* MOOS community, to which the appcast viewer is connected. This may be done with the `pShare` utility. The uField Toolbox also provides a set of MOOS utilities to facilitate this kind of arrangement.

1.5.4 The AppCast Data Structure

An example appcast is shown in Figure 6. It has four distinct parts:

- **config warnings:** Configuration warnings are typically generated during the application's `OnStartUp()` routine.
- **run warnings:** Run warnings may be generated any time during the execution of the application.
- **general messages:** A list of app developer-formatted strings having whatever the application developer thought would best constitute a succinct meaningful status report.
- **events:** A set of time-stamped events. Exactly what constitutes an event is determined by the application developer.

The screenshot shows a terminal window titled "uProcessWatch gilda". At the top right, it says "1/1 (150)". The window is divided into four main sections:

- Config Warnings:** Contains one warning: "[1 of 1]: Unhandled config line: foobar=abracadabra".
- Run Warnings:** Contains one warning: "[1]: Process [pNodeReporter] is missing".
- General messages:** A table showing processes being watched:

ProcName	Watch Reason	Status
pBasicContactMgr	ANT	DB OK
pHelmIpvP	ANT WATCH	DB OK
pHostInfo	ANT	DB OK
pLogger	ANT	DB OK
pMarinePID	ANT WATCH	DB OK
pNodeReporter	ANT WATCH	DB MISSING
pShare	ANT	DB OK
uFldMessageHandler	ANT	DB OK
uFldNodeBroker	ANT	DB OK
uSimMarine	ANT WATCH	DB OK
- Events:** A list of recent events:
 - [120.15]: PROC_WATCH_EVENT: Process [pNodeReporter] is missing.
 - [0.00]: Noted to be present: [pShare]
 - [0.00]: Noted to be present: [pLogger]
 - [0.00]: Noted to be present: [pBasicContactMgr]
 - [0.00]: Noted to be present: [pHostInfo]
 - [0.00]: Noted to be present: [uFldNodeBroker]
 - [0.00]: Noted to be present: [uFldMessageHandler]
 - [0.00]: Noted to be present: [uSimMarine]

Figure 6: An appcast consists of four main parts: (1) configuration warnings, (2) run-time warnings, (3) general status report messages, and (4) run-time events. The bar at the top of the figure is rendered in red due to the presence of a run-time warning. The "1/1" on this line indicates one configuration warning and one run-time warning. The "(150)" on this line indicates that it is iteration #150 for this application. This image is a screen shot take from the uMACView utility described later.

For any given appcast, all fields are optional. Indeed, often an appcast will be devoid of any configuration or run warnings. It is also not uncommon for an application not to have a notion of an event.

For reasons explained later, an appcast instance is typically created once upon application startup. The block of general messages is cleared and overwritten each time an appcast is generated. Run warnings and events are added any time during the application operation, but are limited in amount (first-in-first-out/FIFO). This is done to ensure against unbounded growth of the appcast message, and relieve the app developer from addressing the logic of bounded message growth. Configuration warnings are unbounded however since they are only generated at startup time and are typically bounded from above by the number of application configuration lines.

1.5.5 A Preview of AppCast Viewing Utilities

An appcast viewer is primarily a utility for viewing appcasts, rendering an appcast to look something like that shown in Figure 6. It also does a couple other important things. First, it provides a mechanism to allow the user to navigate between incoming appcasts from multiple vehicles, each with multiple applications. Second, it implements, under the hood, a protocol between the appcast viewer utility and the applications, to ensure on-demand appcasting. The latter will be discussed later in Section 11.10.

The `uMAC` utility shown on the left in Figure 7 is run from a terminal window. The `uMACView` utility shown on the right is a GUI with a bit more capable interface, allowing the user to see all vehicles, all apps for a chosen vehicle, and the appcast for a single chosen application. The advantage of the `uMAC` utility however is that it may be launched remotely after logging in to a vehicle that may otherwise be unresponsive and in need of some debugging.

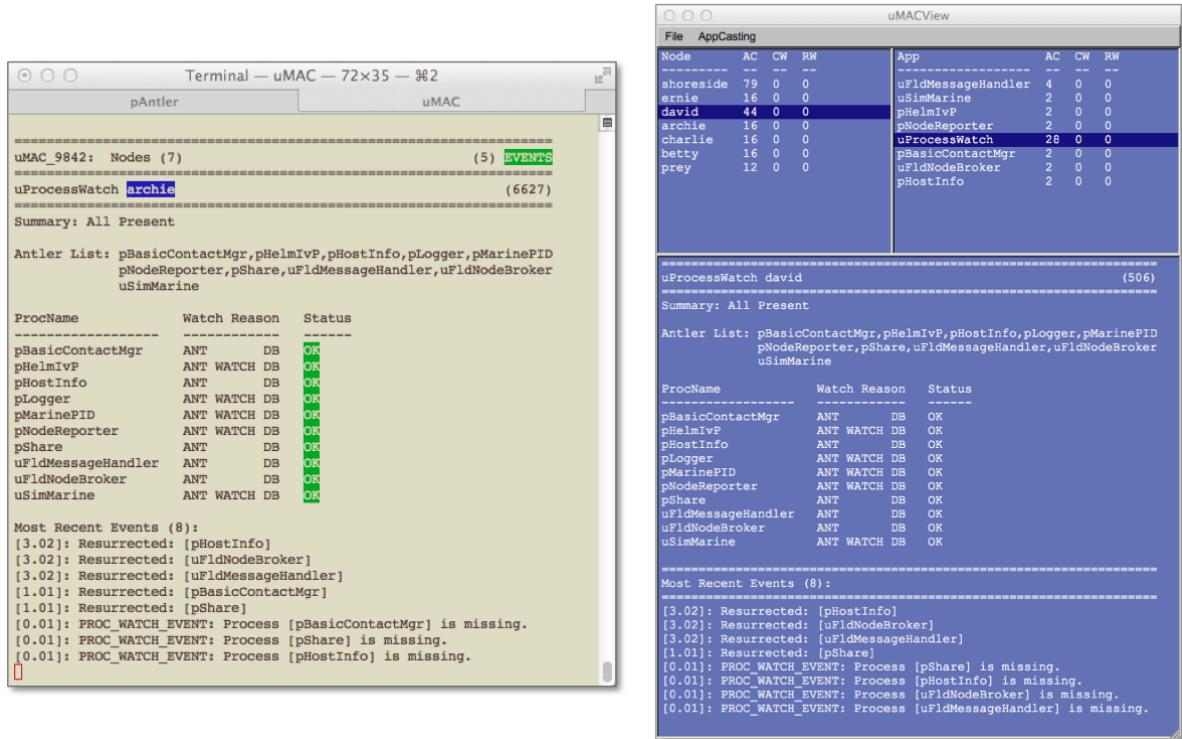


Figure 7: Two appcast viewer utilities: On the left is the terminal-based `uMAC` utility. On the right is the GUI-based `uMACView` utility. Both provide access to the same information. While the latter has a bit nicer user interface, the former may be run remotely while ssh'ing into a fielded vehicle.

While the `uMAC` and `uMACView` utilities are completely stand-alone and do not assume the use of any other tool, a third option exists for users of the `pMarineViewer` tool. This tool has been augmented to support an integrated `uMACView` style interface into the same single window as shown in Figure 8. The appcasting interface may be toggled on and off by simply hitting the 'a' key. The user may also specify the mode upon startup.

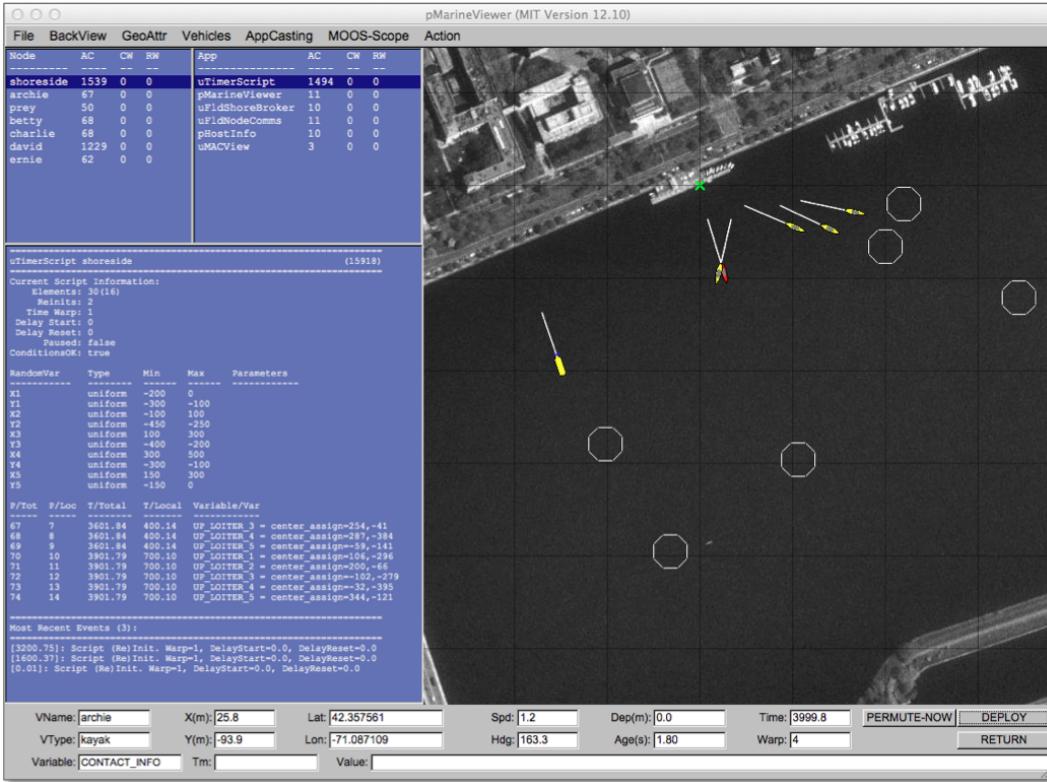


Figure 8: The **pMarineViewer** application has been augmented to support appcast viewing in a separate window pane. The interface is nearly identical to that of the **uMACView** application. The integration is for better convenience to existing **pMarineViewer** users. The appcasting information may be toggled on and off with the 'a' key.

2 pMarineViewer: A GUI for Mission Monitoring and Control

2.1 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 9.

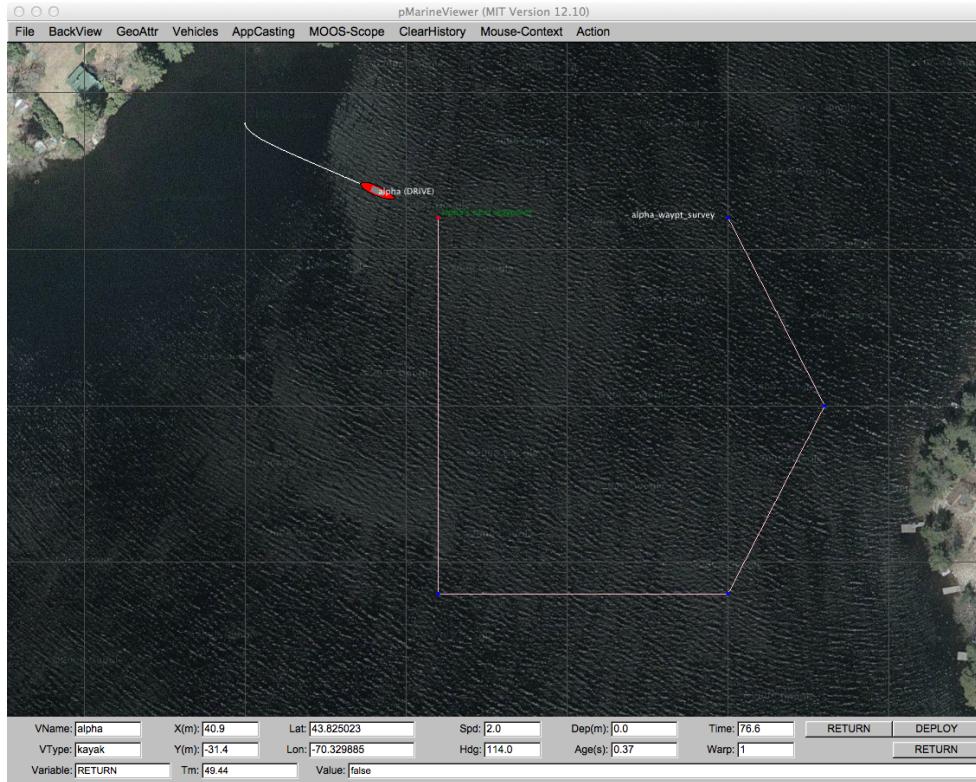


Figure 9: A **pMarineViewer** screen-shot executing a simple one-vehicle mission. The track of the vehicle is shown along with the set of waypoints it will traverse during this mission.

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 2.5.

2.1.1 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

10, 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.

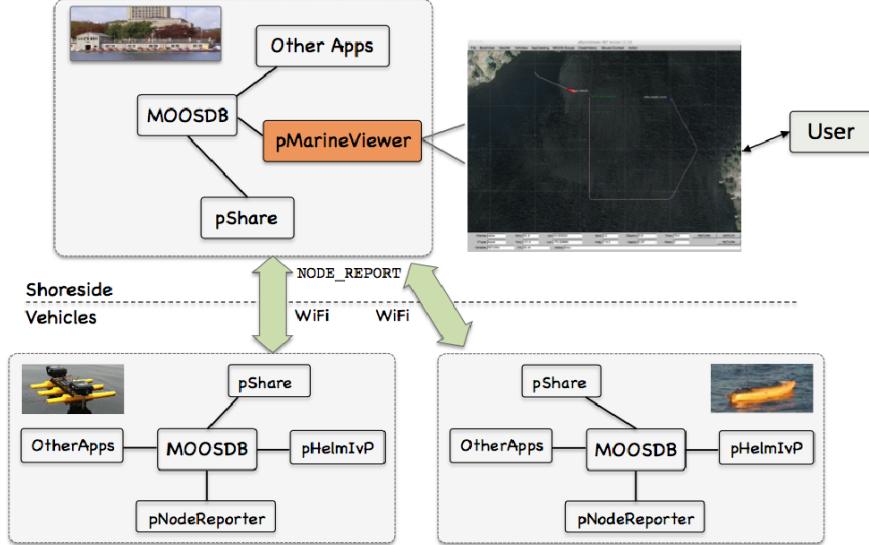


Figure 10: A common usage of the `pMarineViewer` is to have it running in a local `MOOSDB` community while receiving node reports on vehicle poise from other MOOS communities running on either real or simulated vehicles. The vehicles can also send messages with certain geometric information such as polygons 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 2.3.

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 2.1.5.

2.1.2 Description of the pMarineViewer GUI Interface

The viewable area of the GUI has three parts as shown in Figure 11 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.

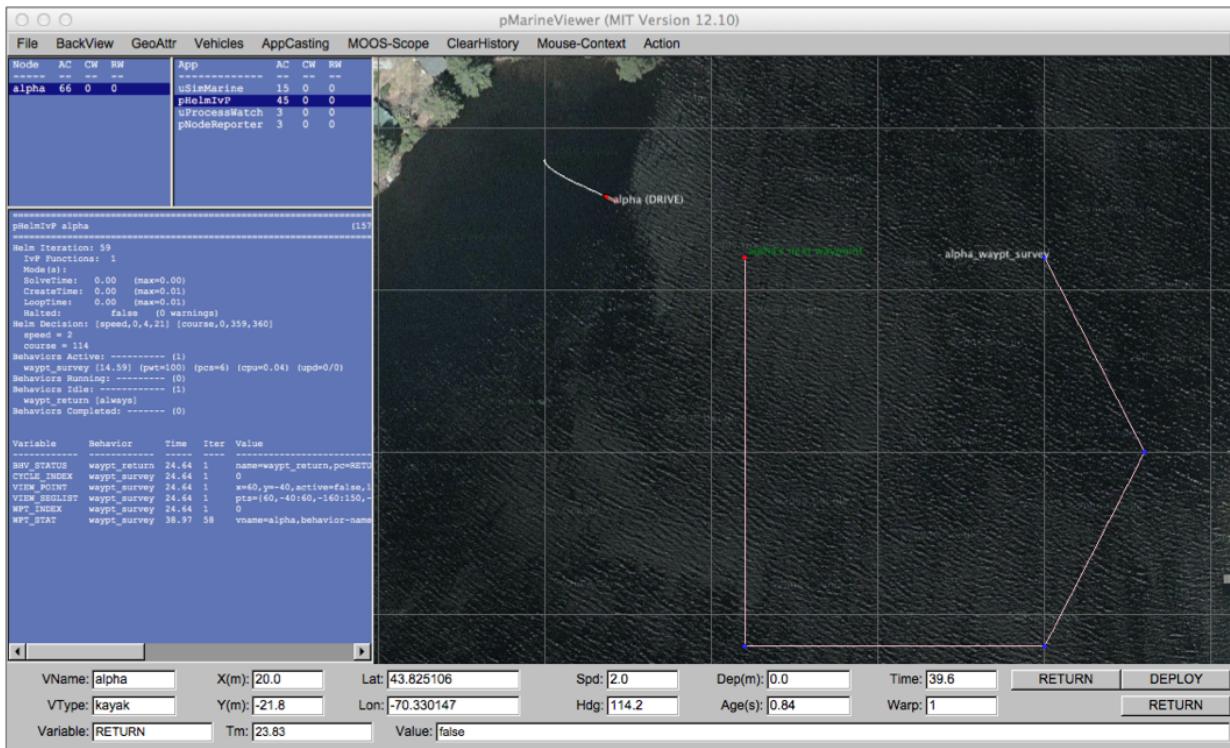


Figure 11: A screen shot of the `pMarineViewer` application running the alpha example mission. The position, heading, speed and other information related to the vehicle is reflected in the data fields at the bottom of the viewer.

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 **pMarineViewer** was 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 2.9 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 2.9 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 2.6 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.

2.1.3 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 11. 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.

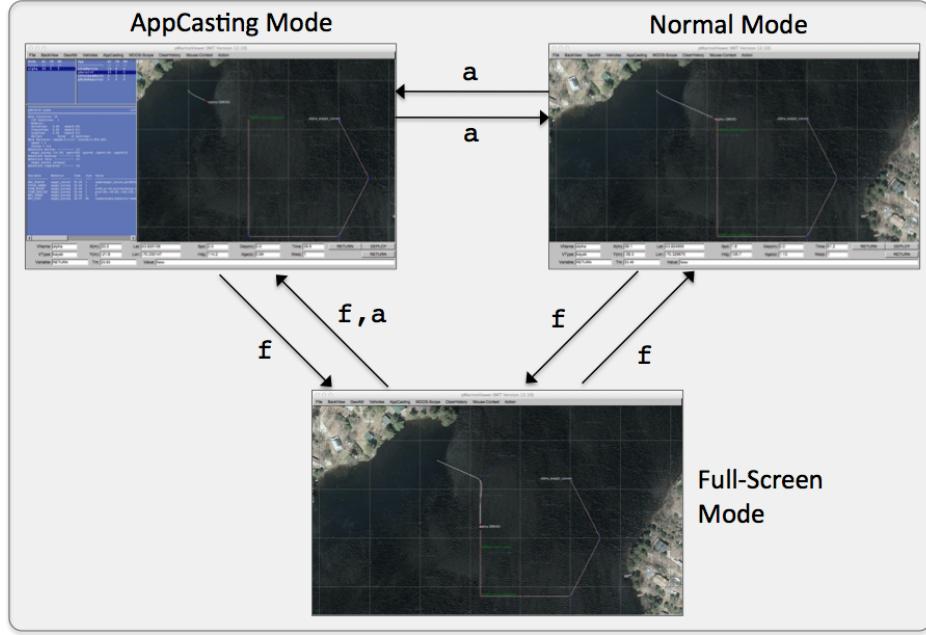


Figure 12: Three viewing modes are supported by pMarineViewer. The *normal* mode, the *appcasting* mode which renders appcast output from any connected vehicle, or the *full-screen* mode to maximize viewing of the operation area and vehicles. The modes may be toggled with the hot-keys shown. When typing '*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.

2.1.4 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.

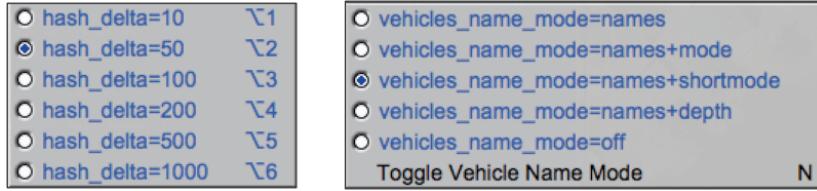


Figure 13: Most configuration parameters may be altered with pull-down menu selections. The radio-button shows the present parameter value and its neighbors show other legal settings. The text of the pull-down menu selection may be placed verbatim in the `.moos` configuration block to determine the setting 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 13, or a hot key for toggling between options as on the right in the figure.

2.1.5 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 2.7 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 2.7.

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 2.8.

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 9, and 11. These buttons, for example, are configured as follows:

```
button_one = DEPLOY # DEPLOY=true
button_one = MOOS_MANUAL_OVERRIDE=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.

2.2 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 14. 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`.

2.2.1 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 2.4.

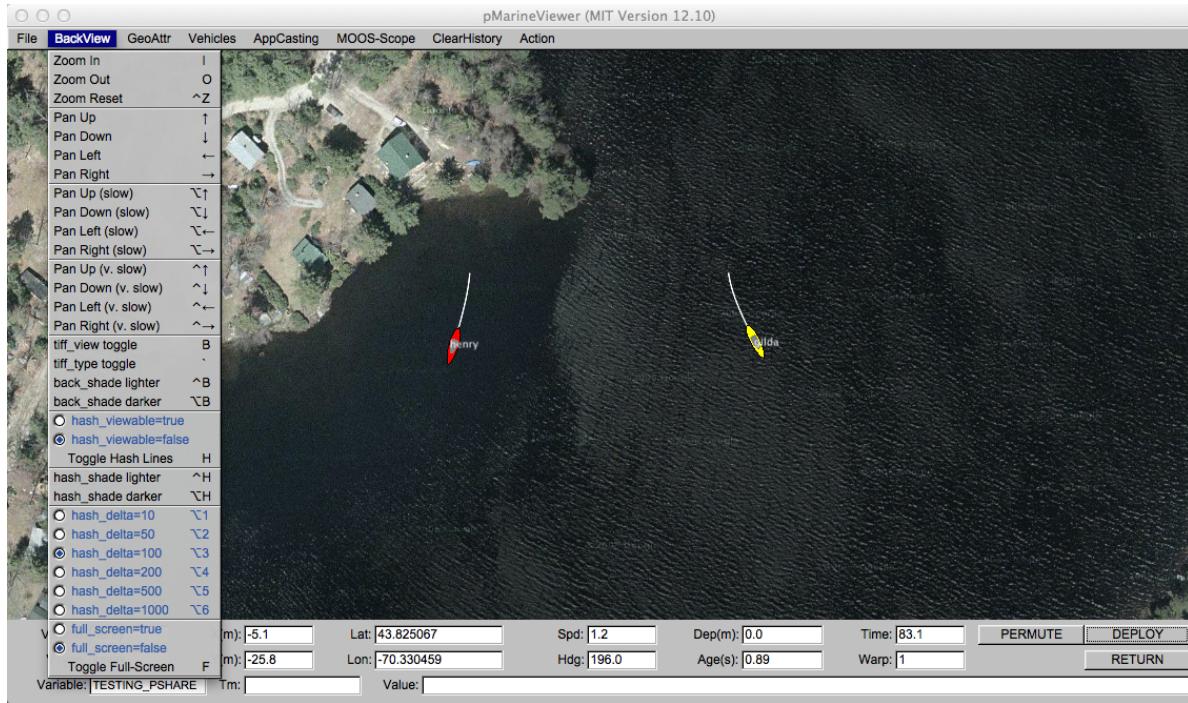


Figure 14: **The BackView menu:** This pull-down menu lists the options, with hot-keys, for affecting rendering aspects of the geo-display background.

2.2.2 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 1.

Listing 2.1: 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 15 with two images of Dabob Bay in Washington State. Both image files were created from resources at www.maps.google.com.

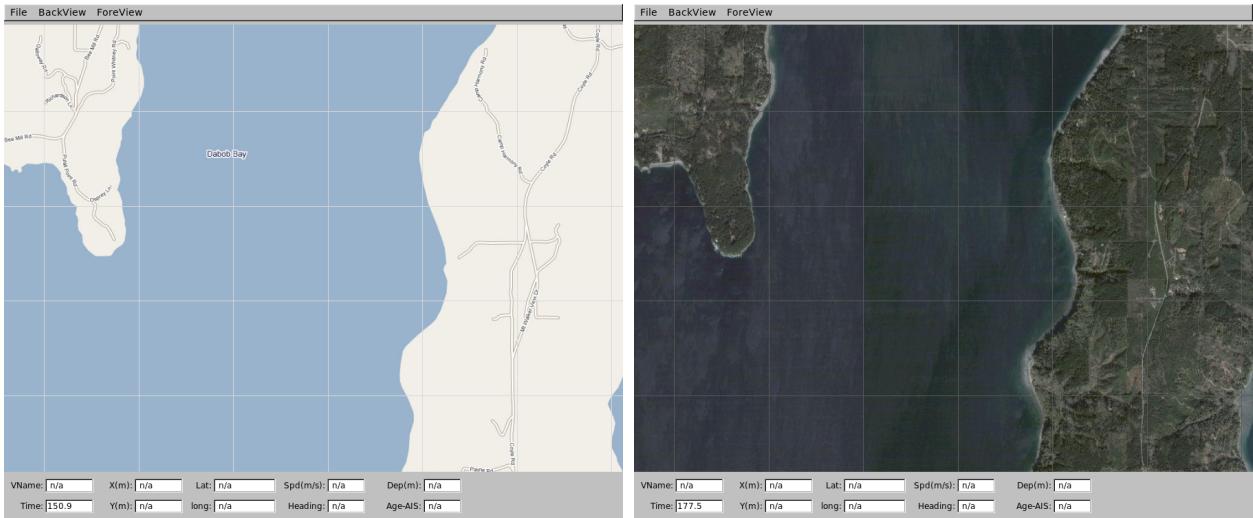


Figure 15: Dual background geo images: Two images loaded for use in the geo display mode of **pMarineViewer**. The user can toggle between both as desired during operation.

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.

2.2.3 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.

2.2.4 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 replaced 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 is false. Full-screen mode is useful when running simulations while connected to a low-resolution projector for example.

2.3 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 2.3.5) are labeled points the user may drop on the viewing area for reference or mission planning

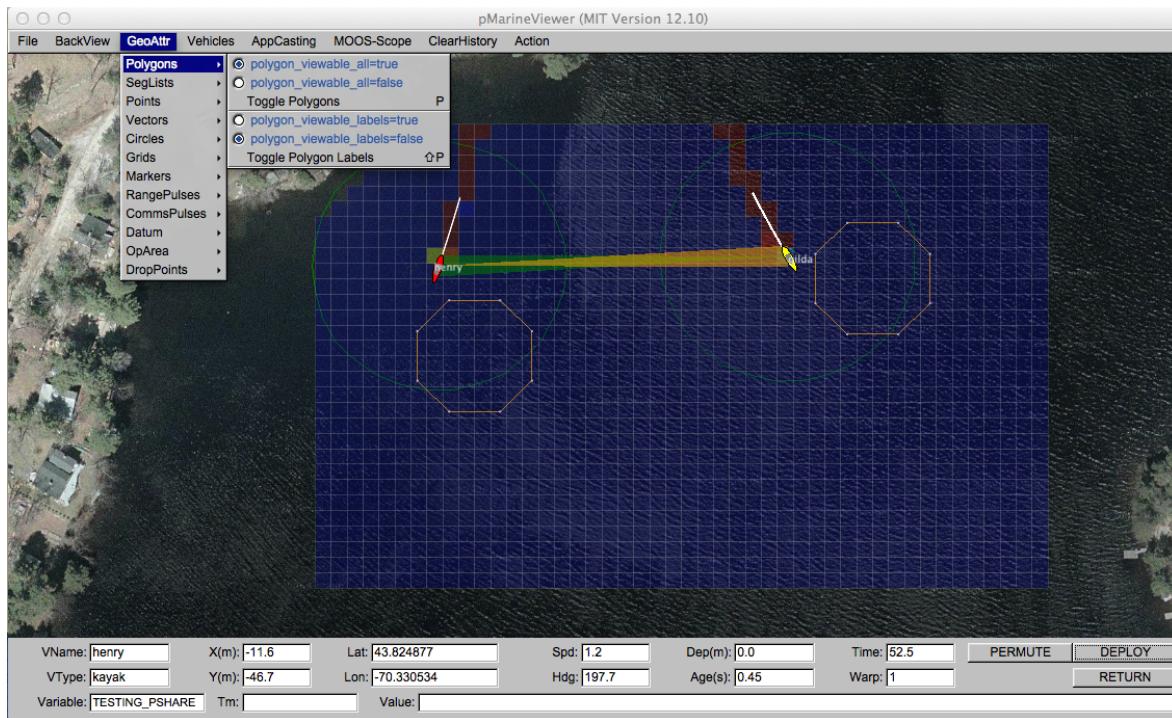


Figure 16: **The "GeoAttr" menu:** This pull-down menu lists the options and hot keys for affecting the rendering of geometric objects.

The possible parameters settings for rendering the geometric objects received by `pMarineViewer` via MOOS mail is provided in Section 2.10.2.

2.3.1 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 9, 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 9. 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 **pMarineViewer**. 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 10.

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.

2.3.2 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 17. 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.

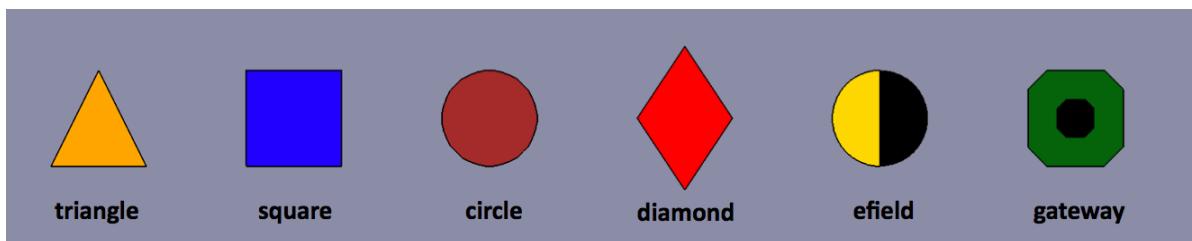


Figure 17: **Markers:** Types of markers known to the `pMarineViewer`.

The color parameter is optional and markers have the default colors shown in Figure 17. 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.

2.3.3 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 18. 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.

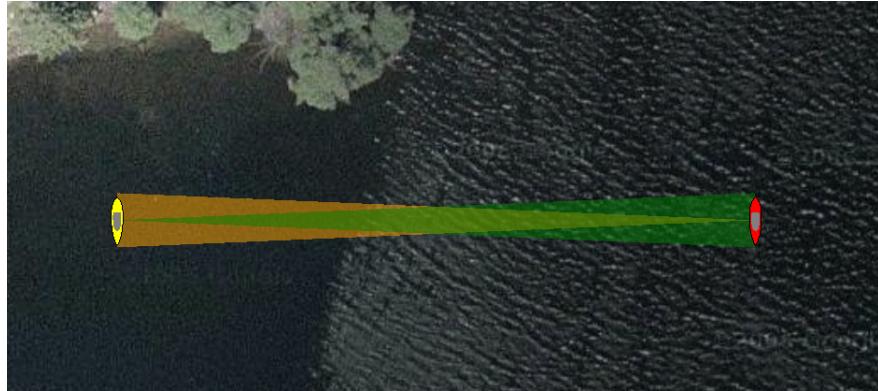


Figure 18: **Comms Pulses:** A comms pulse directionally renders communication between vehicles. Here each vehicle is communicating with the other, and two different colored pulses are rendered.

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 is worth examining an example comms pulse message:

```
VIEW_COMM_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_pulse.set_beam_width(7);
my_pulse.set_fill(0.35);
my_pulse.set_color("fill", "white");
string str = my_pulse.get_spec();
m_Comms.Notify("VIEW_COMM_PULSE", str);
```

The white comms pulse shown in Figure 19 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.

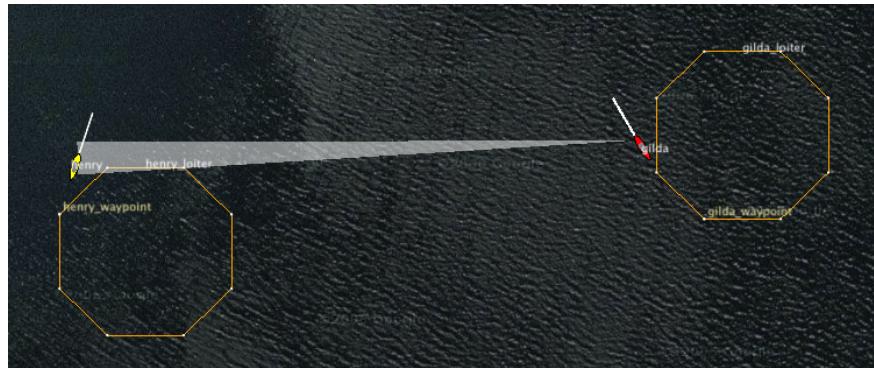


Figure 19: **Comms Pulses for Messaging:** In this figure the white comms pulse indicates that a message is being sent from one vehicle to another, via `uFldNodeComms`.

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 '`o`' 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`.

2.3.4 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 20. Here the pulse is shown over three successive times.

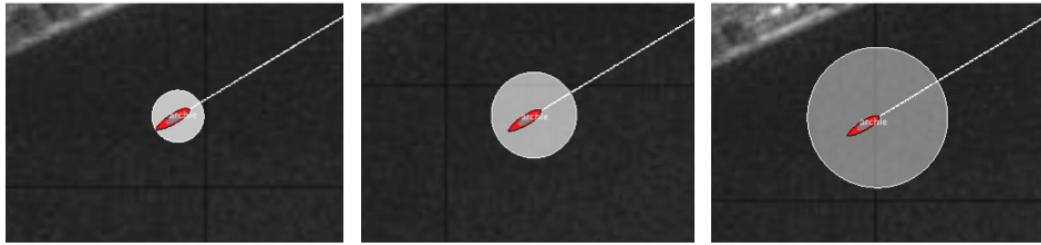


Figure 20: **Comms Pulses:** A comms pulse directionally renders communication between vehicles. Here each vehicle is communicating with the other, and two different colored pulses are rendered.

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 is 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_pulse.set_edge_size(1);
my_pulse.set_radius(50);
my_pulse.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);
```

2.3.5 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 released. 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 21.

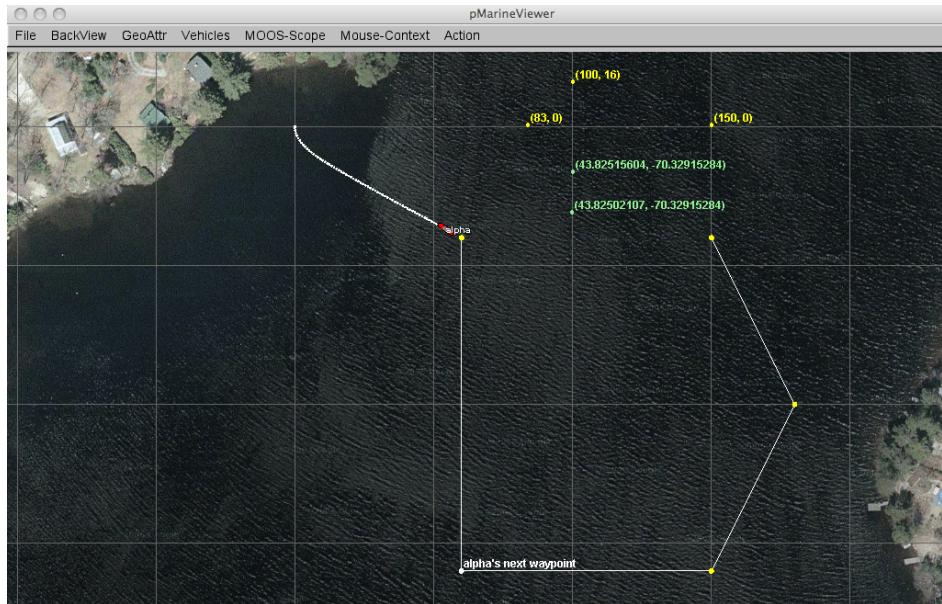


Figure 21: Drop points: A user may leave drop points with coordinates on the geo portion of the **pMarineViewer** window. The points may be rendered in local coordinates or in lat/lon coordinates. The points are added by clicking the left mouse button while holding the **SHIFT** key or **CTRL** key. 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.

2.4 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 22. The very first option is to turn on or off the rendering of all vehicles. This 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.

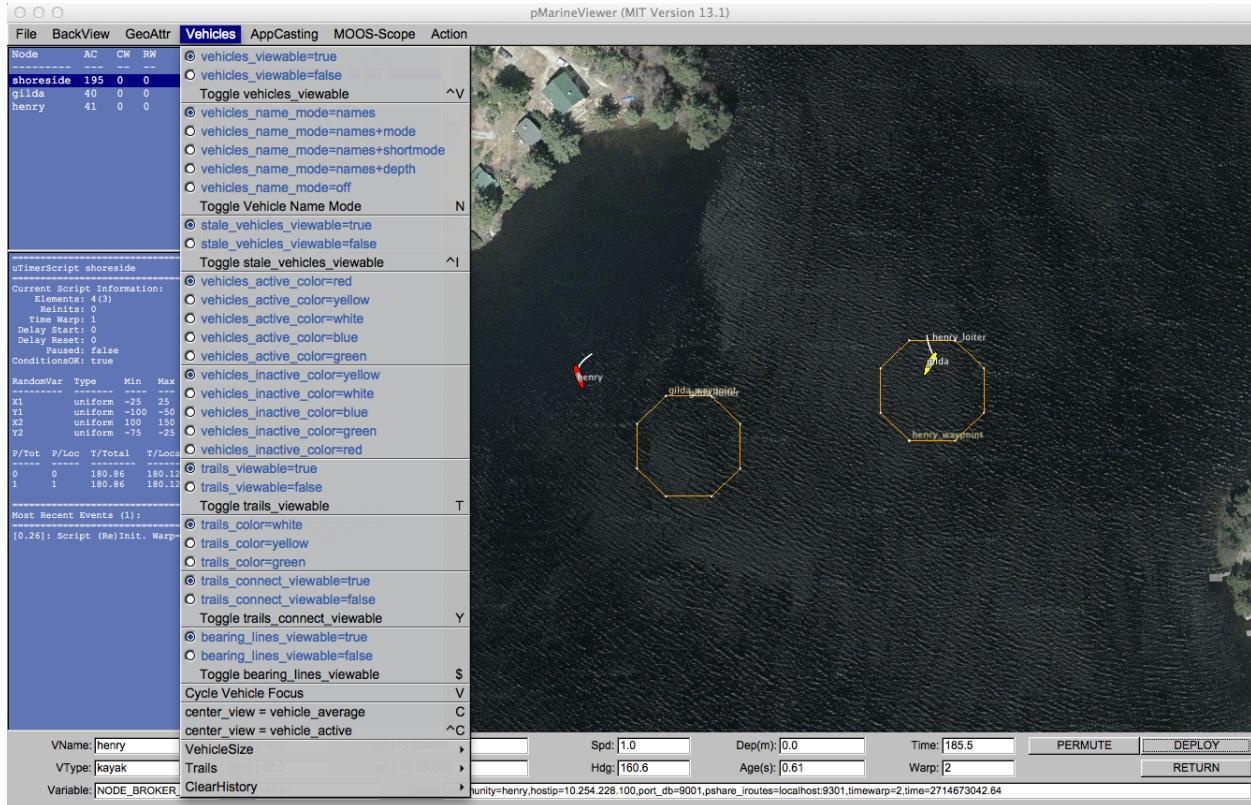


Figure 22: **The Vehicles menu:** This pull-down menu lists the options, with hot-keys, for affecting rendering vehicles and vehicle track history.

2.4.1 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 B with the `vehicles.name_color` configuration parameter.

2.4.2 Dealing with Stale Vehicles

A stale vehicle is one who has not been heard from for a some 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 2.4.6, 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 (30 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 (30 by default), the vehicle may no longer rendered and removed from the appcasting pane.

A few features of this policy are configurable through the mission configuration file. The duration of time after which a vehicle is reported as stale may be changed from its default of 30 seconds with the `stale_report_thresh` parameter. The duration of time after which a vehicle is removed may be changed from its default of 30 seconds with the `stale_remove_thresh` parameter.

Stale vehicles are handled a bit differently when running in simulation and when running vehicles in the field. The difference between the two is determined by the MOOS time warp. Although it's possible to simulate with a time warp of one, here a time warp of one is interpreted as running physical vehicles. Simulated vehicles will be automatically removed from the viewer after `stale_report_thresh + stale_remove_thresh` seconds. When running actual vehicles, stale vehicles must be explicitly removed using the `alt+'k'` key to remove all stale vehicles, or `ctrl+'k'` key to remove the currently selected vehicle in the appcast pane.

2.4.3 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 23.

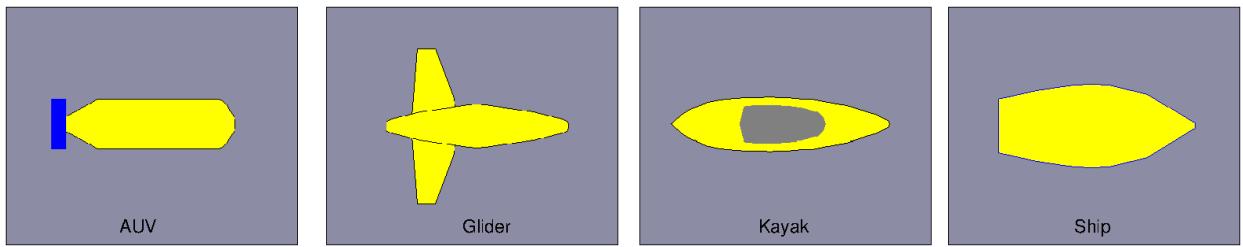


Figure 23: **Vehicles:** Types of vehicle shapes supported by pMarineViewer.

The default shape for an unknown vehicle type is currently set to be the shape "ship".

2.4.4 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 B are acceptable.

2.4.5 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.

2.4.6 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 B. 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 parameter `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.

2.5 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 11. The focus here is on the AppCast menu items and their effect on rendering to the user.

2.5.1 Turning On and Off AppCast Viewing

The AppCast pull-down menu, shown in Figure 24 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.

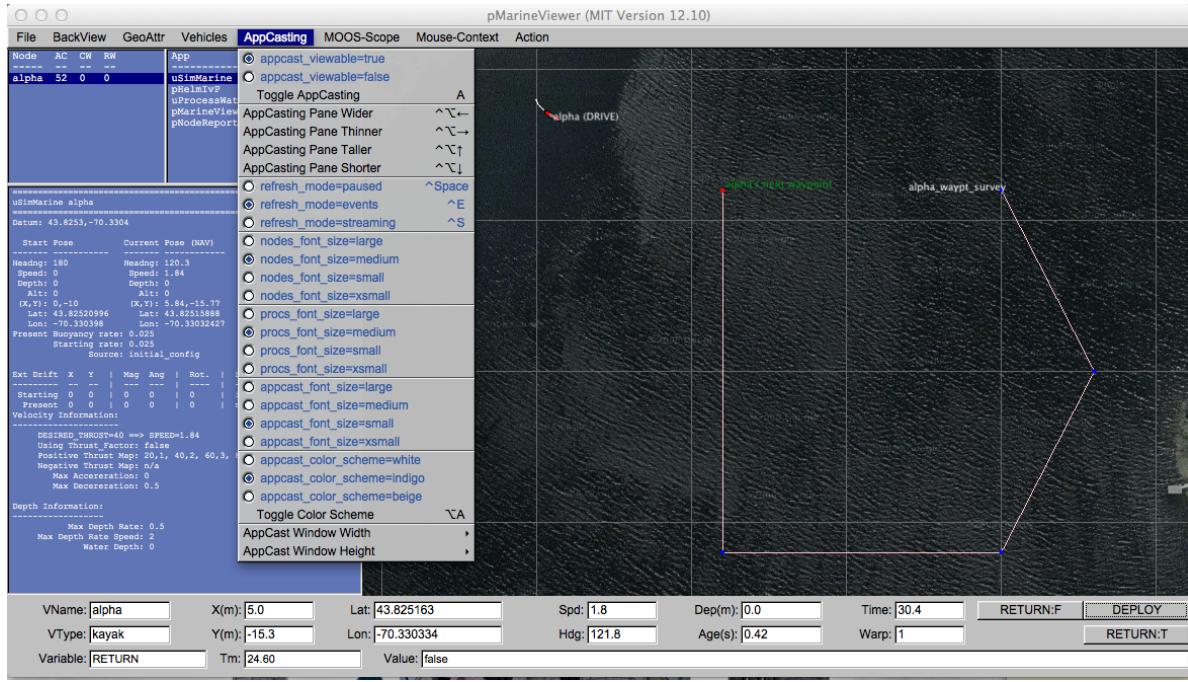


Figure 24: **The AppCast menu:** This pull-down menu lists the options, with hot-keys, for affecting rendering aspects of the appcast panels, and policy for soliciting appcasts from known vehicles and applications.

2.5.2 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 used. 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.

2.5.3 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 11.10.4, 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=logging`.

2.5.4 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.

2.5.5 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 24 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.

2.6 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 25.

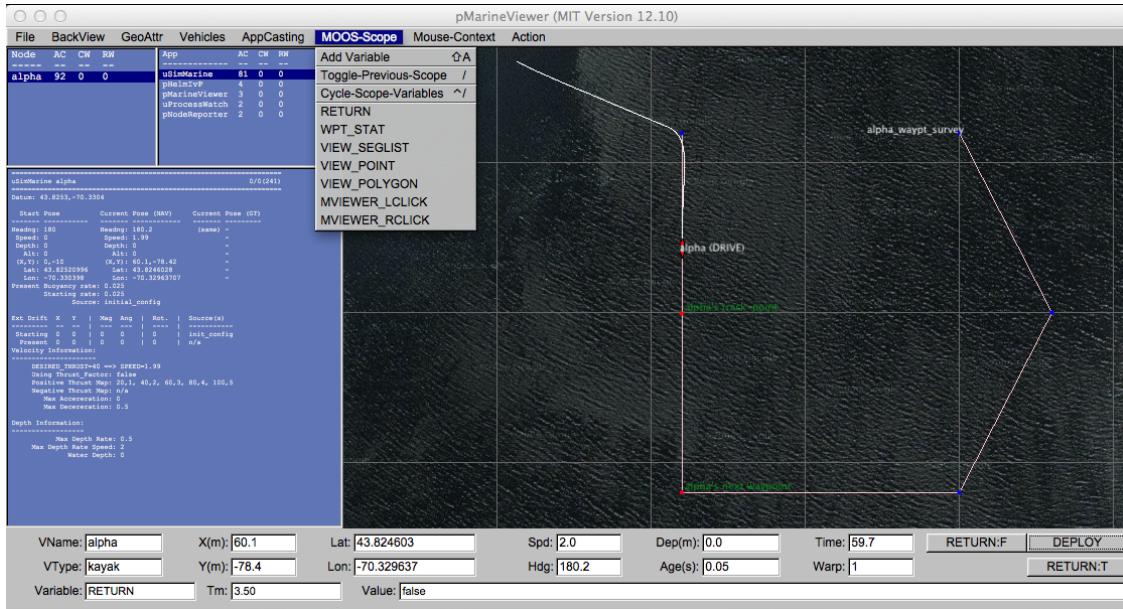


Figure 25: **The Scope Menu:** This pull-down menu allows the user to adjust which pre-configured MOOS variable is to be scoped, or to add a new variable to the scope list.

The three fields show (a) the variable name, (b) the last time it 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 25):

```
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.

2.7 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 2.1.5), 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 26.

```
action = menu_key=deploy # DEPLOY = true # RETURN = false
action+ = menu_key=deploy # MOOS_MANUAL_OVERRIDE=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.

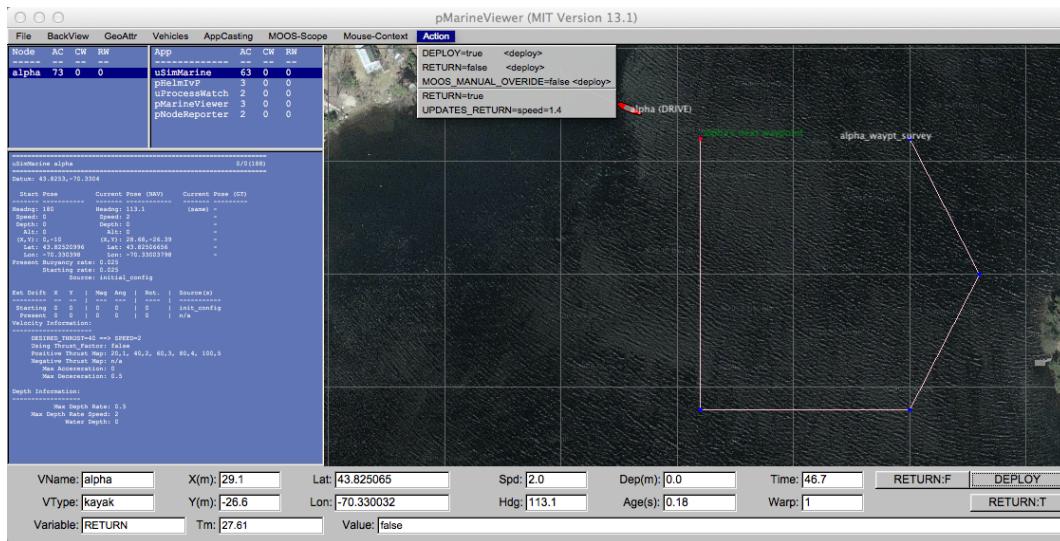


Figure 26: **The Action menu:** The variable value pairs on each menu item may be selected for poking or writing the `MOOSDB`. The three variable-value pairs above the menu divider will be poked in unison when any of the three are chosen, because they were configured with the same key, `<deploy>`, shown 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.

2.8 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.

2.8.1 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.

2.8.2 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 27.

```
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>`.

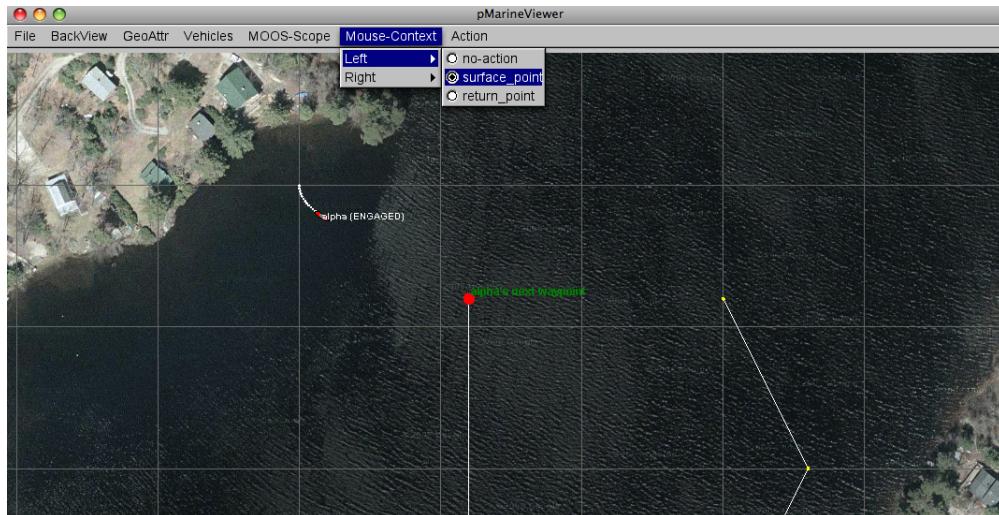


Figure 27: **The Mouse-Context menu:** Keywords selected from this menu will determine which groups of MOOS variables will be poked to the `MOOSDB` on left or mouse clicks. The variable values may have information embedded indicating the position of the mouse in the operation area at the time of the click.

2.9 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 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 28. 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.

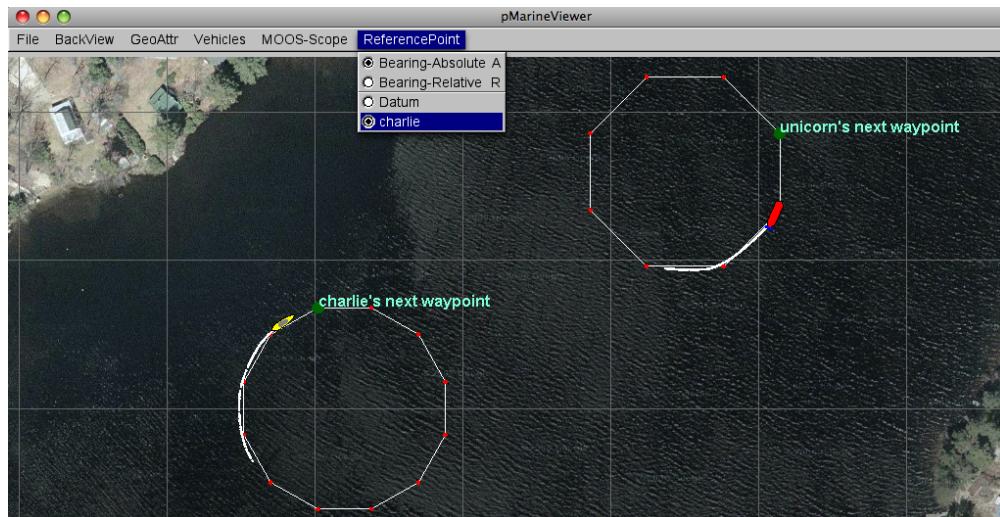


Figure 28: **The Reference-Point menu:** This pull-down menu of the `pMarineViewer` lists the options for selecting a reference point. The reference point determines the values for the `Range` and `Bearing` fields in the viewer for the active vehicle. When the reference point is a vehicle with known heading, the user also may select whether the Bearing is the relative bearing or absolute bearing.

Mini Exercise #1: 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).

2.10 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 14, the menu item `full_screen=true` may also be set in the `pMarineViewer` configuration block verbatim with `full_screen=true`.

2.10.1 Configuration Parameters for the BackView Menu

The parameters in Listing 2 relate to the BackView menu described more fully in Section 2.2. 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 2.2: 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 2.2.4.
- `hash_delta`: Sets the hash line spacing. Legal values: 50, 100, 200, 500, 1000. The default is 100. Section 2.2.3.

hash_shade: Shade of hash marks. Legal value range: [0, 1]. Zero is black, one is white. Section [2.2.3](#).
hash_viewable: If true, hash lines are rendered over the op area. Legal values: true, false. The default is false. Section [2.2.3](#).
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 [2.2.2](#).
tiff_file: Filename of a tiff file background image. Section [2.2.2](#).
tiff_file.b: Filename of another tiff file background image. Section [2.2.2](#).
tiff_type: Use the first tiff image if set to true. Legal values: true, false. The default is true. Section [2.2.2](#).
tiff_viewable: Use the tiff background image if set to true. Otherwise a gray screen is used as a background. Legal values: true, false. The default is true. Section [2.2.2](#).
view_center: Sets the center of the viewing image in x,y local coordinates. Legal values: (double,double). The default is (0,0).

2.10.2 Configuration Parameters for the GeoAttributes Menu

The parameters in Listing 3 relate to the GeoAttributes pull-down menu described more fully in Section [2.3](#). 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 2.3: 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 [2.3.1](#).
circle_viewable_labels: If false, circle labels are suppressed from rendering. Legal values: true, false. The default is true. Section [2.3.1](#).
comms_pulse_viewable_all: If false, comms pulses are suppressed from rendering. Legal values: true, false. The default is true. Section [2.3.3](#).
datum_viewable: If false, the datum is suppressed from rendering. Legal values: true, false. The default is true. Sections [2.2.2](#) and [2.3](#).
datum_color: The color used for rendering the datum. Legal values: Any color listed in the Colors Appendix. The default is red. Sections [2.2.2](#) and [2.3](#).
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 [2.2.2](#) and [2.3](#).
drop_point_viewable_all: If false, drop points are suppressed from rendering. Legal values: true, false. The default is true. Section [2.3.5](#).

<code>drop_point_coords:</code>	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 2.3.5 .
<code>drop_point_vertex_size:</code>	The size of the point used to render a drop point. Legal values: Integers in the range [1, 10]. The default is 2. Section 2.3.5 .
<code>grid_viewable_all:</code>	If false, grids are suppressed from rendering. Legal values: true, false. The default is true.
<code>grid_viewable_labels:</code>	If false, grid labels are suppressed from rendering. Legal values: true, false. The default is true.
<code>grid_viewable_opaqueness:</code>	The degree to which grid renderings are opaque. Legal range: [0, 1]. The default is 0.3.
<code>marker</code>	A marker may be stated in the configuration file with the same format of the <code>VIEW_MARKER</code> message. Section 2.3.2 .
<code>marker_scale:</code>	The scale applied to marker renderings. Legal range: [0.1, 100]. The default is 1.0. Section 2.3.2 .
<code>marker_viewable_all:</code>	If false, markers are suppressed from rendering. Legal values: true, false. The default is true. Section 2.3.2 .
<code>marker_edge_width:</code>	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 2.3.2 .
<code>marker_viewable_labels:</code>	If false, marker labels are suppressed from rendering. Legal values: true, false. The default is true. Section 2.3.2 .
<code>oparea_viewable_all:</code>	If false, oparea lines are suppressed from rendering. Legal values: true, false. The default is true.
<code>oparea_viewable_labels:</code>	If false, oparea label is suppressed from rendering. Legal values: true, false. The default is true.
<code>point_viewable_all:</code>	If false, points are suppressed from rendering. Legal values: true, false. The default is true. Section 2.3.1 .
<code>point_viewable_labels:</code>	If false, point labels are suppressed from rendering. Legal values: true, false. The default is true. Section 2.3.1 .
<code>polygon_viewable_all:</code>	If false, polygons are suppressed from rendering. Legal values: true, false. The default is true. Section 2.3.1 .
<code>polygon_viewable_labels:</code>	If false, polygon labels are suppressed from rendering. Legal values: true, false. The default is true. Section 2.3.1 .
<code>range_pulse_viewable_all:</code>	If false, range pulses are suppressed from rendering. Legal values: true, false. The default is true. Section 2.3.3 .
<code>seglist_viewable_all:</code>	If false, seglists are suppressed from rendering. Legal values: true, false. The default is true. Section 2.3.1 .
<code>seglist_viewable_labels:</code>	If false, seglist labels are suppressed from rendering. Legal values: true, false. The default is true. Section 2.3.1 .
<code>vector_viewable_all:</code>	If false, vectors are suppressed from rendering. Legal values: true, false. The default is true. Section 2.3.1 .

`vector_viewable_labels`: If false, vector labels are suppressed from rendering. Legal values: true, false. The default is true. Section 2.3.1.

2.10.3 Configuration Parameters for the Vehicles Menu

The parameters in Listing 4 relate to the Vehicles pull-down menu described more fully in Section 2.4. 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 2.4: 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 2.4.5.

`stale_remove_thresh`: Number of seconds after a stale vehicle has been detetedected before being removed. When time warp is one, vehicles are *not* automatically removed at all, and this number is meaningless. Legal values: Any non-negative number. The default is 30. Section 2.4.2.

`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 2.4.2.

`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 2.4.6.

`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 2.4.6.

`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 2.4.6.

`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 2.4.6.

`trails_viewable`: If false, vehicle trails are suppressed from rendering. Legal values: true, false. The default is true. Section 2.4.6.

`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 2.4.4.

`vehicles_inactive_color`: The color of inactive vehicles. Legal values: Any color listed in the Colors Appendix. The default is yellow. Section 2.4.4.

<code>vehicles_shape_scale</code> :	The scale factor applied to vehicle size rendering. Legal values in the range: [0.1, 100]. The default is 1.0. Section 2.4.3 .
<code>vehicles_name_mode</code> :	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 2.4.1 .
<code>vehicles_name_color</code> :	Sets the color for rendering the vehicle label. Legal values are any color in Appendix B . The default is white. Section 2.4.4 .
<code>vehicles_viewable</code> :	If false, vehicles are suppressed from rendering. Legal values: true, false. The default is true. Section 2.4 .

2.10.4 Configuration Parameters for the AppCast Menu

The parameters in Listing 5 relate to the AppCast pull-down menu described more fully in Section [2.5](#). 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 2.5: Configuration Parameters for pMarineViewer AppCast Pull-Down Menu.

<code>appcast_color_scheme</code> :	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 2.5.5 .
<code>appcast_font_size</code> :	The font size uses in the <i>appcast</i> pane of the set of appcasting panes. Legal values: large, medium, small, xsmall. The default is small. Section 2.5.4 .
<code>appcast_height</code> :	The height of the appcasting bottom pane as a percentage of the total <i>pMarineViewer</i> window height. Legal values: [30, 35, 40, 45,..., 85, 90]. The default is 75. Section 2.5.2 .
<code>appcast_viewable</code> :	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 2.5.1 .
<code>appcast_width</code> :	The width of the appcasting panes as a percentage of the total <i>pMarineViewer</i> window width. Legal values: [20, 25, 30, 35,..., 65, 70]. The default is 30. Section 2.5.2 .
<code>nodes_font_size</code> :	the font size uses in the <i>nodes</i> pane of the set of appcasting panes. Legal values: large, medium, small, xsmall. The default is medium. Section 2.5.4 .
<code>procs_font_size</code> :	The font size uses in the <i>procs</i> pane of the set of appcasting panes. Legal values: large, medium, small, xsmall. The default is medium. Section 2.5.4 .
<code>refresh_mode</code> :	Determines the manner in which appcast requests are sent to apps. Legal values: paused, events, streaming. The default is events. Section 2.5.3 .

2.10.5 Configuration Parameters for the Scope, MouseContext and Action Menus

Listing 2.6: Configuration Parameters the Scope, MouseContext and Action Menus.

<code>scope</code> :	A comma separated list of MOOS variables to scope. Section 2.6.
<code>oparea</code> :	A specification of the operation area boundary for optionally rendering.
<code>button_one</code> :	A configurable command and control button. Section 2.1.5.
<code>button_two</code> :	A configurable command and control button. Section 2.1.5.
<code>button_three</code> :	A configurable command and control button. Section 2.1.5.
<code>button_four</code> :	A configurable command and control button. Section 2.1.5.
<code>action</code> :	A MOOS variable-value pair for posting, available under the Action pull-down menu. Section 2.7.
<code>left_context</code> :	Allows the custom configuration of left mouse click context. Section 2.8.
<code>right_context</code> :	Allows the custom configuration of right mouse click context. Section 2.8.
<code>lclick_ix_start</code> :	Starting index for the left mouse index macro. Section 2.8.
<code>rclick_ix_start</code> :	Starting index for the right mouse index macro. Section 2.8.

2.11 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
```

2.11.1 Variables Published by pMarineViewer

It is possible to configure `pMarineViewer` to poke the `MOOSDB` via either the Action pull-down menu (Section 2.7), or via configurable GUI buttons (Section 2.1.5). It may also publish to the `MOOSDB` variables configured to mouse clicks (Section 2.8). 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 11.10.4.
- `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 `pHelmIpv` 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 2.8.

- **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 2.2.2.

2.11.2 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 11.10.4.
- **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.
- **PHI_HOST_INFO**: A string representing the detected IP address, published by `pHostInfo`. Used for augmenting the `pMarineViewer` title bar with the current IP address.
- **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

3 uHelmScope: Scoping on the IvP Helm

3.1 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 1 is used for explaining the `uHelmScope` fields.

Listing 3.1: 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.

3.2 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 1, 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 `uHelmScope` is not part of the appcast report posted by the helm.

3.2.1 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`.

3.2.2 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 $\{0, 0.25, 0.5, \dots, 3.75, 4.0\}$. 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.

3.2.3 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 were 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.

3.3 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 1 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 1 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 5. 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 1, 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`'.

3.4 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.

3.5 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 1.

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 2.

Listing 3.2: 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 3, and may be obtained from the command line by invoking:

```
$ uHelmScope --help or -h
```

Listing 3.3: 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 irrelevant.
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`.

3.6 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, omitting 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:

```
IVPHELM_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.

3.7 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 5, 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 3.4: 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 '`o`' key. Legal values: true, false. The default is true. Section 3.4.
- `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 3.3.
- `display_virgins`: If true, variables in the MOOS scope section of the report 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 3.3.
- `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 3.3.

An example configuration file may be obtained from the command line with:

```
$ uHelmScope --example or -e
```

This will show the output shown in Listing 5 below.

Listing 3.5: 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 3.5. A parameter setting in the MOOS configuration block will take precedence over a command line switch.

3.8 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 or -i
```

3.8.1 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 11.10.4.
- [IVPHELM_REJOURNAL](#): A request to the helm to rejournal its summary output. Section 3.6.

3.8.2 Variables Subscribed for by uHelmScope

- [APPCAST_REQ](#): A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 11.10.4.

- <USER-DEFINED>: Variables identified for scoping by the user in the `uHelmScope` will be subscribed for. See Section 3.3.
- <HELM-DEFINED>: As described in Section 3.3, 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 3.6.
- `IVPHELM_SUMMARY`: A comprehensive summary of the helm status including behavior status summaries and most recent helm decision. See Section 3.6.
- `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 3.6.
- `IVPHELM_DOMAIN`: The specification of the IvP Domain presently used by the helm. See Section 3.6.
- `IVPHELM_MODESET`: A description of the helm's hierarchical mode specification. See Section 3.6.
- `IVPHELM_STATE`: A short description of the helm state: either STANDBY, PARK, DRIVE, or DISABLED. See Section 3.6.

4 pNodeReporter: Summarizing a Node's Position and Status

4.1 Overview

The `pNodeReporter` MOOS application runs on each vehicle (real or simulated) and generates node-reports (as a proxy for AIS reports) for sharing between vehicles, depicted in Figure 29. The process serves one primary function - it repeatedly gathers local platform information and navigation data and creates an AIS like report in the form of the MOOS variable `NODE_REPORT_LOCAL`. The `NODE_REPORT` messages are communicated between the vehicles and the shore or shipside command and control through an inter-MOOSDB communications process such as pShare or via acoustic modem. Since a node or platform may both generate and receive reports, the locally generated reports are labeled with the `_LOCAL` suffix and bridged to the outside communities without the suffix. This is to ensure that processes running locally may easily distinguish between locally generated and externally generated reports.

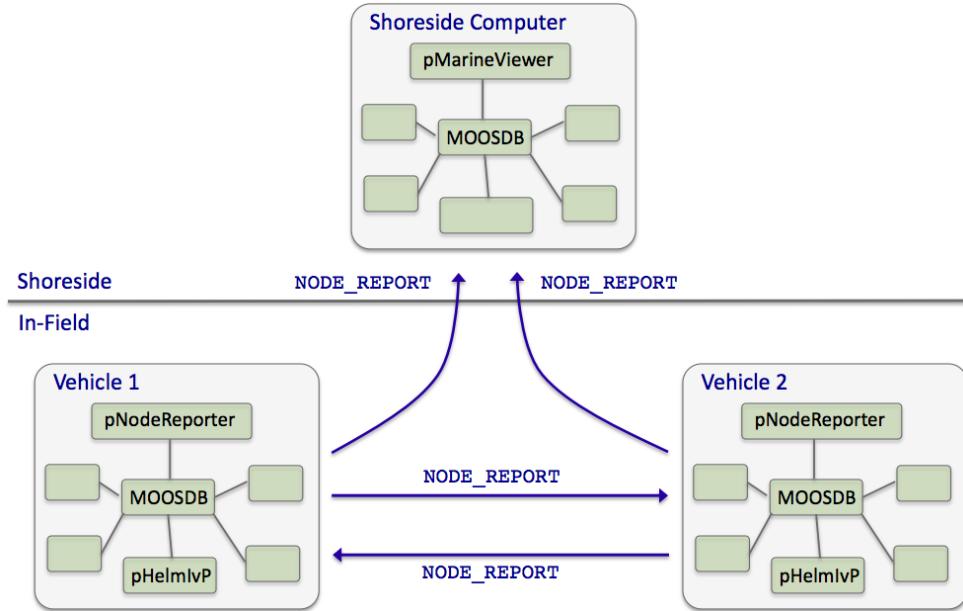


Figure 29: **Typical pNodeReporter usage:** The `pNodeReporter` application is typically used with pShare or acoustic modems to share node summaries between vehicles and to a shoreside command-and-control GUI.

To generate the local report, `pNodeReporter` registers for the local `NAV_*` vehicle navigation data and creates a report in the form of a single string posted to the variable `NODE_REPORT_LOCAL`. An example of this variable is given in below in Section 4.2.1. The `pMarineViewer` and `pHelmIVP` applications are two modules that consume and parse the incoming `NODE_REPORT` messages.

The `pNodeReporter` utility may also publish a second report, the `PLATFORM_REPORT`. While the `NODE_REPORT` summary consists of an immutable set of data fields described later in this section, the `PLATFORM_REPORT` consists of data fields configured by the user and may therefore vary widely across applications. The user may also configure the frequency in which components of the `PLATFORM_REPORT`

are posted within the report.

4.2 Using pNodeReporter

4.2.1 Overview Node Report Components

The primary output of `pNodeReporter` is the node report string. It is a comma-separated list of key-value pairs. The order of the pairs is not significant. The following is an example report:

```
NODE_REPORT_LOCAL = "NAME=alpha,TYPE=UUV,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=MODE@ACTIVE:LOITERING"
```

The `TIME` reflects the Coordinated Universal Time as indicated by the system clock running on the machine where the MOOSDB is running. Speed is given in meters per second, heading is in degrees in the range [0, 360), depth is in meters, and the local x-y coordinates are also in meters. The source of information for these fields is the `NAV_*` navigation MOOS variables such as `NAV_SPEED`. The report also contains several components describing characteristics of the physical platform, and the state of the IvP Helm, described next.

If desired, `pNodeReporter` may be configured to use a different variable than `NODE_REPORT_LOCAL` for its node reports, by setting the configuration parameter `node_report_output` to `MY_REPORT` for example. Most applications that subscribe to node reports, subscribe to two variables, `NODE_REPORT_LOCAL` and `NODE_REPORT`. This is because node reports are meant to be bridged to other MOOS communities (typically with `pShare` but not necessarily). A node report should be broadcast only from the community that generated the report. In practice, to ensure that node reports that arrive in one community are not then sent out to other communities, the node reports generated locally have the `_LOCAL` suffix, and when they are sent to other communities they are sent to arrive with the new variable name, minus the suffix.

4.2.2 Helm Characteristics

The node report contains one field regarding the current mode of the helm, `MODE`. Typically the `pNodeReporter` and `pHelmIvP` applications are running on the same platform, connected to the same MOOSDB. When the helm is running, but disengaged, i.e., in manual override mode, the `MODE` field in the node report simply reads "`MODE=DISENGAGED`". When or if the helm is detected to be not running, the field reads "`MODE=NOHELM-SECS`", where `SECS` is the number of seconds since the last time `pNodeReporter` detected the presence of the helm, or "`MODE=NOHELM-EVER`" if no helm presence has ever been detected since `pNodeReporter` has been launched.

How does `pNodeReporter` know about the health or status of the helm? It subscribes to two MOOS variables published by the helm, `IVPHELM_STATE` and `IVPHELM_SUMMARY`. These are described more fully in [3], but below are typical example values:

```
IVPHELM_STATE    = "DRIVE"  
  
IVPHELM_SUMMARY = "iter=72,ofnum=1,warnings=0,time=127349406.22,solve_time=0.00,  
create_time=0.00,loop_time=0.00,var=course:209.0,var=speed:1.2,
```

```

halted=false,running_bhvs=none,modes=MODE@ACTIVE:LOITERING,
active_bhvs=loiter$17.8$100.00$9$0.04$0/0,completed_bhvs=none
idle_bhvs=waypt_return$17.8$0/0:station-keep$17.8$n/a

```

The `IVPHELM_STATE` variable is published on each iteration of the `pHelmIvP` process regardless of whether the helm is in manual override ("PARK") mode or not, and regardless of whether the value of this variable has changed between iterations. It is considered the "heartbeat" of the helm. This is the variable monitored by `pNodeReporter` to determine whether a "NOHELM" message is warranted. By default, a period of five seconds is used as a threshold for triggering a "NOHELM" warning. This value may be changed by setting the `nohelm_threshold` configuration parameter.

When the helm is indeed engaged, i.e., not in manual override mode, the value of `IVPHELM_STATE` posting simply reads "DRIVE", but the helm further publishes the `IVPHELM_SUMMARY` variable similar to the above example. If the user has chosen to configure the helm using hierarchical mode declarations (as described in [3]), the `IVPHELM_SUMMARY` posting will include a component such as "`modes=MODE@ACTIVE:LOITERING`" as above. This value is then included in the node report by `pNodeReporter`. If the helm is not configured with hierarchical mode declarations, the node report simply reports "MODE=DRIVE".

4.2.3 Platform Characteristics

The node report contains three fields regarding the platform characteristics, NAME, TYPE, and LENGTH. The name of the platform is equivalent to the name of the MOOS community within which `pNodeReporter` is running. The MOOS community is declared as a global MOOS parameter (outside any given process' configuration block) in the `.moos` mission file. The TYPE and LENGTH parameters are set in the `pNodeReporter` configuration block. They may alternatively derive their values from a MOOS variable posted elsewhere by another process. The user may configure `pNodeReporter` to use this external source by naming the MOOS variables with the `platform_length_src` and `platform_type_src` parameters. If both the source and explicit values are set, as for example:

```

platform_length      = 12           // meters
platform_length_src = SYSTEM_LENGTH // A MOOS Variable

```

then the explicit length of 12 would be used only if the MOOS variable `SYSTEM_LENGTH` remained unwritten to by any other MOOS application connected to the MOOSDB. The platform length and type may be used by other platforms as a parameter affecting collision avoidance algorithms and protocol. They are also used in the `pMarineViewer` application to allow the proper platform icon to be displayed at the proper scale.

If the platform type is known, but no information about the platform length is known, certain rough default values may be used if the platform type matches one of the following: "kayak" maps to 4 meters, "uuv" maps to 4 meters, "auv" maps to 4 meters, "ship" maps to 18 meters, "glider" maps to 3 meters.

4.2.4 Dealing with Local versus Global Coordinates

A primary component of the node report is the current position of the vehicle. The `pNodeReporter` application subscribes for the following MOOS variables to garner this information: NAV_X, NAV_Y in

local coordinates, and the pair `NAV_LAT`, `NAV_LONG` in global coordinates. These two pairs should be consistent, but what if they aren't? And what if `pNodeReporter` is receiving mail for one pair but not the other? Three distinct policy choices are supported:

- The default policy: node reports include exactly what is given. If `NAV_X` and `NAV_Y` are being received only, then there will be no entry in the node report for global coordinates, and vice versa. If both pairs are being received, then both pairs are reported. No attempt is made to check or ensure that they are consistent. This is the default policy, equivalent to the configuration `cross_fill_policy=literal`.
- If one of the two pairs is not being received, `pNodeReporter` will fill in the missing pair from the other. This policy can be chosen with the configuration `cross_fill_policy=fill-empty`.
- If one of the two pairs has been received more recently, the older pair is updated by converting from the other pair. The older pair may also be in a state where it has never been received. This policy can be chosen with the configuration `cross_fill_policy=fill-latest`.

4.2.5 Processing Alternate Navigation Solutions

Under normal circumstances, node reports are generated reflecting the current navigation solution as defined by the incoming `NAV_*` variables. The `pNodeReporter` application can handle the case where the vehicle also publishes an alternate navigation solution, as defined by a sister set of incoming MOOS variables separate from the `NAV_*` variables. In this case `pNodeReporter` will monitor both sets of variables and may generate *two* node reports on each iteration. The following two configuration parameters are needed to activate this capability:

```
alt_nav_prefix = <prefix>      // example: NAV_GT_
alt_nav_name   = <node-name>    // example: _GT
```

The configuration parameter, `alt_nav_prefix`, names a prefix for the alternate incoming navigation variables. For example, `alt_nav_prefix=NAV_GT_` would result in `pNodeReporter` subscribing for `NAV_GT_X`, `NAV_GT_Y` and so on. A separate vehicle state would be maintained internally based on this alternate set of navigation information and a second node report would be generated.

A second node report would be published under the same MOOS variable, `NODE_REPORT_LOCAL`, but the NAME component of the report would be distinct based on the value provided in the `alt_nav_name` parameter. If a name is provided that does not begin with an underscore character, that name is used. If the name does begin with an underscore, the name used in the report is the otherwise configured name of the vehicle plus the suffix.

4.3 The Optional Blackout Interval Option

Under normal circumstances, the `pNodeReporter` application will post a node report once per iteration, the gap between postings being determined solely by the `app_tick` parameter (Figure 30). However, there are times when it is desirable to add an artificial delay between postings. Node reports are typically only useful as information sent to another node, or to a shoreside computer rendering fielded vehicles, and there are often dropped node report messages due to the uncertain nature of communications in the field, whether it be acoustic communications, wifi, or satellite link.

Applications receiving node reports usually implement provisions that take dropped messages into account. A collision-avoidance or formation-following behavior, or a contact manager, may extrapolate a contact position from its last received position and trajectory. A shoreside command-and-control GUI such as `pMarineViewer` may render an interpolation of vehicle positions between node reports. To test the robustness of applications needing to deal with dropped messages, a way of simulating the dropped messages is desired. One way is to add this to the simulation version of whatever communications medium is being used. For example, there is an acoustic communications simulator where the dropping of messages may be simulated, where the probability of a drop may even be tied to the range between vehicles. Another way is to simply simulate the dropped message at the source, by adding delay to the posting of reports by `pNodeReporter`.

By setting the `blackout_interval` parameter, `pNodeReporter` may be configured to ensure that a node report is not posted until *at least* the duration specified by this parameter has elapsed, as shown in Figure 31.

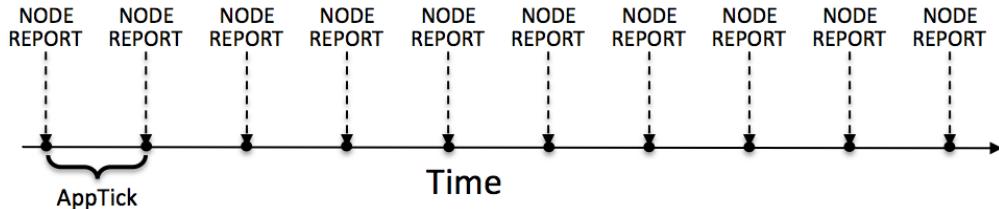


Figure 30: **Normal schedule of node report postings:** The `pNodeReporter` application will post node reports once per application iteration. The duration of time between postings is directly tied to the frequency at which `pNodeReporter` is configured to run, as set by the standard MOOS `AppTick` parameter.

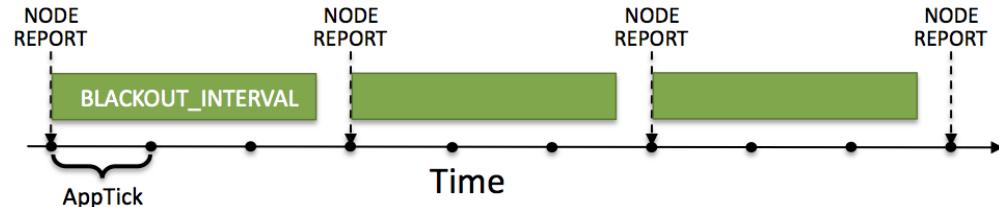


Figure 31: **The optional blackout interval parameter:** The schedule of node report postings may be altered by setting the `BLACKOUT_INTERVAL` parameter. Reports will not be posted until at least the time specified by the blackout interval has elapsed since the previous posting.

An element of unpredictability may be added by specifying a value for the `blackout_variance` parameter. This parameter is given in seconds and defines an interval $[-t, t]$ from which a value is chosen with uniform probability, to be added to the duration of the blackout interval. This variation is re-calculated after each interval determination. The idea is depicted in Figure 32.

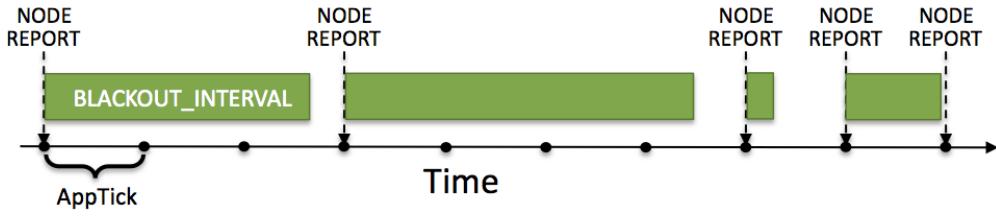


Figure 32: **Blackout intervals with varying duration:** The duration of a blackout interval may be configured to vary randomly within a user-specified range, specified in the `blackout_variance` parameter.

Message dropping is typically tied semi-predictably to characteristics of the environment, such as range between nodes, water temperature or platform depth, an so on. This method of simulating dropped messages captures none of that. It is however simple and allows for easily proceeding with the testing of applications that need to deal with the dropped messages.

4.4 Configuration Parameters for pNodeReporter

The following parameters are defined for `pNodeReporter`. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so in parentheses.

Listing 4.1: Configuration Parameters for pNodeReporter.

- `alt_nav_prefix`: Source for processing alternate nav reports. Section 4.2.5.
- `alt_nav_name`: Node name in posting alternate nav reports. Section 4.2.5.
- `blackout_interval`: Minimum duration, in seconds, between reports (0). Section 4.3.
- `blackout_variance`: Variance in uniformly random blackout duration. Legal values: any non-negative value. The default is zero. Section 4.3.
- `cross_fill_policy`: Policy for handling local versus global nav reports ("literal"). Section 4.2.4.
- `node_report_output`: MOOS variable used for the node report (`NODE_REPORT_LOCAL`). Section 4.2.1.
- `nohelm_threshold`: Seconds after which a quiet helm is reported as AWOL. Legal values: any non-negative value. The default is 5 seconds. Section 4.2.2.
- `platform_length`: The reported length of the platform in meters. Legal values: any non-negative value. The default is zero. Section 4.2.3.
- `plat_report_output`: The Platform report MOOS variable. Legal values: conventions for MOOS variable names. The default is `PLATFORM_REPORT_LOCAL`. Section 4.6.
- `plat_report_input`: A component of the optional platform report. Section 4.6.
- `platform_type`: The reported type of the platform. Legal values: any string. The default is "unknown". Section 4.2.3.

An Example MOOS Configuration Block

An example MOOS configuration block is provided in Listing 2 below. To see an example MOOS configuration block from the console, enter the following:

```
$ pNodeReporter --example or -e
```

This will show the output shown in Listing 2 below.

Listing 4.2: Example configuration of pNodeReporter.

```
1 =====
2 pNodeReporter Example MOOS Configuration
3 =====
4 Blue lines: Default configuration
5
6 ProcessConfig = pNodeReporter
7 {
8     AppTick    = 4
9     CommsTick = 4
10
11    // Configure key aspects of the node
12    platform_type      = glider    // or {uuv,aув,ship,kayak}
13    platform_length    = 8         // meters. Range [0,inf)
14
15    // Configure optional blackout functionality
16    blackout_interval  = 0         // seconds. Range [0,inf)
17
18    // Configure the optional platform report summary
19    plat_report_input  = COMPASS_HEADING, gap=1
20    plat_report_input  = GPS_SAT, gap=5
21    plat_report_input  = WIFI_QUALITY, gap=1
22    plat_report_output = PLATFORM_REPORT_LOCAL
23
24    // Configure the MOOS variable containing the node report
25    node_report_output = NODE_REPORT_LOCAL
26
27    // Threshold for conveying an absence of the helm
28    nohelm_threshold   = 5         // seconds
29
30    // Policy for filling in missing lat/lон from x/y or v.versa
31    crossfill_policy   = literal  // or {fill-empty,use-latest}
32
33    // Configure monitor/reporting of dual nav solution
34    alt_nav_prefix     = NAV_GT
35    alt_nav_name       = _GT
36 }
```

4.5 Publications and Subscriptions for pNodeReporter

The interface for **pNodeReporter**, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pNodeReporter --interface or -i
```

4.5.1 Variables Published by pNodeReporter

The primary output of pNodeReporter to the MOOSDB is the node report and the optional platform report:

- **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.
- **NODE_REPORT_LOCAL**: Primary summary of the node's navigation and helm status. Section 4.2.1.
- **PLATFORM_REPORT_LOCAL**: Optional summary of certain platform characteristics. Section 4.2.3.

4.5.2 Variables Subscribed for by pNodeReporter

Variables subscribed for by **pNodeReporter** are summarized below. A more detailed description of each variable follows. In addition to these variables, any MOOS variable that the user requests to be included in the optional **PLATFORM_REPORT** will also be automatically subscribed for.

- **APPCAST_REQ**: A request to generate and post a new appcast report, with reporting criteria, and expiration. Section 11.10.4.
- **IVPHELM_STATE**: A indicator of the helm state produced by **pHelmIvP**, e.g., "PARK", "DRIVE" "DISABLED", or "STANDBY".
- **IVPHELM_ALLSTOP**: A indicator of the helm allstop produced by **pHelmIvP**, e.g., "clear", or a reason why the vehicle is at zero speed.
- **IVPHELM_SUMMARY**: A summary report produced by the IvP Helm (**pHelmIvP**).
- **NAV_X**: The ownship vehicle position on the *x* axis of local coordinates.
- **NAV_Y**: The ownship vehicle position on the *y* axis of local coordinates.
- **NAV_LAT**: The ownship vehicle position on the *y* axis of global coordinates.
- **NAV_LONG**: The ownship vehicle position on the *x* axis of global coordinates.
- **NAV_HEADING**: The ownship vehicle heading in degrees.
- **NAV_YAW**: The ownship vehicle yaw in radians.
- **NAV_SPEED**: The ownship vehicle speed in meters per second.
- **NAV_DEPTH**: The ownship vehicle depth in meters.

If **pNodeReporter** is configured to handle a second navigation solution as described in Section 4.2.5, the corresponding additional variables described in that section will also be automatically subscribed for.

4.5.3 Command Line Usage of pNodeReporter

The **pNodeReporter** 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. The basic command line usage for the **pNodeReporter** application is the following:

*Listing 4.3: Command line usage for the **pNodeReporter** application.*

```

1 Usage: pNodeReporter file.moos [OPTIONS]
2
3 Options:
4   --alias=<ProcessName>
5     Launch pNodeReporter with the given process name
6     rather than pNodeReporter.
7   --example, -e
8     Display example MOOS configuration block.
9   --help, -h
10    Display this help message.
11   --version,-v
12    Display the release version of pNodeReporter.

```

4.6 The Optional Platform Report Feature

The `pNodeReporter` application allows for the optional reporting of another user-specified list of information. This report is made by posting the `PLATFORM_REPORT_LOCAL` variable. An alternative variable name may be used by setting the `PLAT_REPORT_SUMMARY` configuration parameter. This report may be configured by specifying one or more components in the `pNodeReporter` configuration block, of the following form:

```
plat_report_input = <variable>, gap=<duration>, alias=<variable>
```

If no component is specified, then no platform report will be posted. The `<variable>` element specifies the name of a MOOS variable. This variable will be automatically subscribed for by `pNodeReporter` and included in (not necessarily all) postings of the platform report. If the variable `BODY_TEMP` is specified, a component of the report may contain "`BODY_TEMP=98.6`". An alias for a MOOS variable may be specified. For example, `alias=T`, for the `BODY_TEMP` component would result in "`T=98.6`" in the platform report instead.

How often is the platform report posted? Certainly it will not be posted any more often than the `apptick` parameter allows, but it may be posted far more infrequently depending on the user configuration and how often the values of its components are changing. The platform report is posted only when one or more of its components requires a re-posting. A component requires a re-posting only if (a) its value has changed, *and* (b) the time specified by its gap setting has elapsed since the last platform report that included that component. When a `PLATFORM_REPORT_LOCAL` posting is made, only components that required a posting will be included in the report.

The wide variation in configurations of the platform report allow for reporting information about the node that may be very specific to the platform, not suitable for a general-purpose node report. As an example, consider a situation where a shoreside application is running to monitor the platform's battery level and whether or not the payload compartment has suffered a breach, i.e., the presence of water is detected inside. A platform report could be configured as follows:

```
plat_report_input = ACME_BATT_LEVEL, gap=300, alias=BATTERY_LEVEL
plat_report_input = PAYLOAD_BREACH
```

This would result in an initial posting of:

```

PLATFORM_REPORT_LOCAL = "platform=alpha,utc_time=1273510720.99,BATTERY_LEVEL=97.3,
PAYLOAD_BREACH=false"

```

In this case, the platform uses batteries made by the ACME Battery Company and the interface to the battery monitor happens to publish its value in the variable `ACME_BATT_LEVEL`, and the software on the shoreside that monitors all vehicles in the field accepts the generic variable `BATTERY_LEVEL`, so the alias is used. It is also known that the ACME battery monitor output tends to fluctuate a percentage point or two on each posting, so the platform report is configured to include a battery level component no more than once every five minutes, (`gap=300`). The MOOS process monitoring the indication of a payload breach is known to have few false alarms and to publish its findings in the variable `PAYLOAD_BREACH`. Unlike the battery level which has frequent minor fluctuations and degrades slowly, the detection of a payload breach amounts to the flipping of a Boolean value and needs to be conveyed to the shoreside as quickly as possible. Setting `gap=0`, the default, ensures that a platform report is posted on the very next iteration of `pNodeReporter`, presumably to be read by a MOOS process controlling the platform's outgoing communication mechanism.

4.7 An Example Platform Report Configuration Block for pNodeReporter

Listing 4 below shows an example configuration block for `pNodeReporter` where an extensive platform report is configured to report information about the autonomous kayak platform to support a “kayak dashboard” display running on a shoreside computer. Most of the components in the platform report are specific to the autonomous kayak platform, which is precisely why this information is included in the platform report, and not the node report.

Listing 4.4: An example pNodeReporter configuration block.

```

1 //-----
2 // pNodeReporter config block
3
4 ProcessConfig = pNodeReporter
5 {
6     AppTick = 2
7     CommsTick = 2
8
9     platform_type      = KAYAK
10    platform_length    = 3.5      // Units in meters
11    nohelm_thresh     = 5        // The default
12    blackout_interval = 0        // The default
13    blackout_variance = 0        // The default
14
15    node_report_output = NODE_REPORT_LOCAL      // The default
16    plat_report_output = PLATFORM_REPORT_LOCAL   // The default
17
18    plat_report_input = COMPASS_PITCH, gap=1
19    plat_report_input = COMPASS_HEADING, gap=1
20    plat_report_input = COMPASS_ROLL, gap=1
21    plat_report_input = DB_UPTIME, gap=1
22    plat_report_input = COMPASS_TEMPERATURE, gap=1, alias=COMPASS_TEMP
23    plat_report_input = GPS_MAGNETIC_DECLINATION, gap=10, alias=MAG_DECL

```

```
24     plat_report_input = GPS_SAT, gap=5
25     plat_report_input = DESIRED_RUDDER, gap=0.5
26     plat_report_input = DESIRED_HEADING, gap=0.5
27     plat_report_input = DESIRED_THRUST, gap=0.5
28     plat_report_input = GPS_SPEED, gap=0.5
29     plat_report_input = DESIRED_SPEED, gap=0.5
30     plat_report_input = WIFI_QUALITY, gap=0.5
31     plat_report_input = WIFI_QUALITY, gap=1.0
32     plat_report_input = MOOS_MANUAL_OVERRIDE, gap=1.0
33 }
```

5 uXMS: Scoping the MOOSDB from the Console

5.1 Overview

uXMS is a terminal based MOOS app for scoping the MOOSDB. It has no graphics library build dependencies and is easily launched from the command line to scope on just what the user wants, from everything to just one important variable. For example, typing the following on the command line after say the alpha example mission is launched, will result in Figure 33.

```
$ uXMS alpha.moos NAV_X NAV_Y DEPLOY RETURN IVPHELM_STATE
```

VarName	(S)ource	(T)ime	(C)ommunity	VarValue (SCOPING: EVENTS)
DEPLOY	pHelmIpvP	7.77	alpha	"false"
IVPHELM_STATE	pHelmIpvP	1212.42	alpha	"PARK"
NAV_X	uSimMarine	1212.41	alpha	0
NAV_Y	uSimMarine	1212.41	alpha	-10
RETURN	pHelmIpvP	7.77	alpha	"false"

Figure 33: **A simple scope on five variables with uXMS:** A line is used for each variable showing the variable name, the time of the most recent posting, the source, and the current value.

Scoping on the MOOSDB is a very important tool in the process of development and debugging. The **uXMS** tool has a substantial set of configuration choices for making this job easier by bringing just the right data to the user's attention. The default usage, as shown above, is fairly simple, but there are other options discussed in this section that are worth exploiting by the more experienced user.

- *Use with appcasting:* **uXMS** is appcast enabled meaning its terminal reports may be viewed with tools other than the terminal window. It is possible to configure multiple **uXMS** scopes to automatically launch with a mission and viewer each of them in remote appcast viewing tools. More on this topic in Section 5.9.
- *Scoping on history:* **uXMS** may be configured to scope on the history of a variable to view not just its current state but recent values.
- *Remote low-bandwidth scoping:* **uXMS** can be launched and connected to a remote MOOSDB over a low-bandwidth link, with refresh requests made only on the user's request. **uXMS** can also be on a remote vehicle via an ssh session.
- *Dynamic changes to the scope list:* The set of scoped variables may be altered dynamically by selecting MOOS apps to include or exclude from the scope list.

At any time the user may hit the 'h' key to see a list of help commands.

5.2 The uXMS Refresh Modes

Reports such as the one shown in Figure 33 are generated either automatically or specifically when the user asks for it. The latter is important in situations where bandwidth is low. This feature was the original motivation for developing [uXMS](#). When a new report is sent to the terminal is determined by the *refresh mode*. The three refresh modes are shown in Figure 34 along with the key strokes for switching between modes.

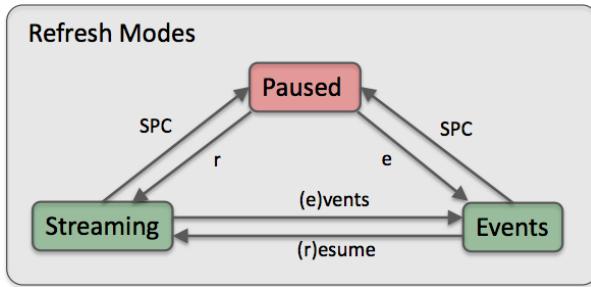


Figure 34: **Refresh Modes:** The uXMS refresh mode determines when a new report is written to the screen. The user may switch between modes with the shown keystrokes.

The refresh mode may be changed by the user as [uXMS](#) is running, or it may be given an initial mode value on startup from the command line with `--mode=paused`, `--mode=streaming`, or `--mode=events`. The latter is the default. It may also be set in the [uXMS](#) configuration block in the mission file with the `refresh_mode` parameter. The current refresh mode is shown in parentheses in the report header as shown in Figure 33 where it is in the *events* refresh mode.

5.2.1 The Streaming Refresh Mode

In the *streaming* refresh mode, a new report is generated and written to stdout on every iteration of the [uXMS](#) application. The frequency is limited from above by the apptick setting in the MOOS configuration block. It is also limited from above by the parameter `term_report_interval`, which is by default 0.6 seconds. Each report written to the terminal will show an incremented counter at the end of the first line, in parentheses. This counter represents the [uXMS](#) iteration counter. This mode may be entered by hitting the '`r`' key, or chosen as the initial refresh mode at startup from the command line with the `--mode=streaming` option.

5.2.2 The Events Refresh Mode

In the *events* refresh mode, the default refresh mode, a new report is generated only when new mail is received for one of the scoped variables. Note this does not necessarily mean that the value of the variable has changed, only that it has been written to again by some process. As with the *streaming* mode, the report frequency is limited by the apptick and the `term_report_interval` setting. This mode is useful in low-bandwidth situations where a user cannot afford the streaming refresh mode, but may be monitoring changes to one or two variables. This mode may be entered by hitting the '`e`' key, or chosen as the initial refresh mode at startup from the command line with the `--mode=events` option.

5.2.3 The Paused Refresh Mode

In the *paused* refresh mode, the report will not be updated until the user specifically requests a new update by hitting the spacebar key. This mode is the preferred mode in low bandwidth situations, and simply as a way of stabilizing the rapid refreshing output of the other modes so one can actually read the output. This mode is entered by the spacebar key and subsequent hits refresh the output once. To launch `uXMS` in the paused mode, use the `--mode=paused` command line switch.

5.3 The uXMS Content Modes

The contents of the `uXMS` report vary between one of a few modes. In the *scoping* mode, a snapshot of a subset of MOOS variables is generated, similar to what is shown in Figure 33. In the *history* mode the recent history of changes to a single MOOS variable is reported.

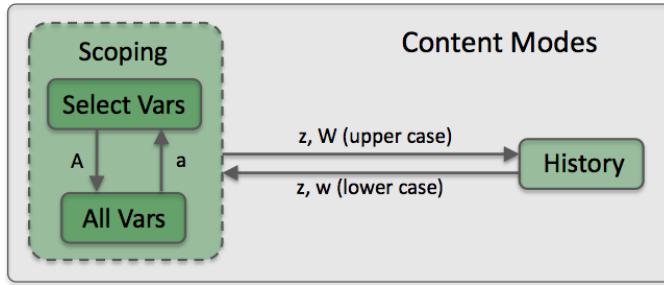


Figure 35: **Content Modes:** The `uXMS` content mode determines what data is included in each new report. The two major modes are the *scoping* and *history* modes. In the former, snapshots of one or more MOOS variables are reported. In the latter, the recent history of a single variable is reported.

5.3.1 The Scoping Content Mode

The *scoping* mode has two sub-modes as shown in Figure 35. In the first sub-mode, the *SelectVars* sub-mode, the only variables shown are the ones the user requested. They are requested on the command-line upon start-up (Section 5.5), or in the `uXMS` configuration block in the `.moos` file provided on startup, or both. One may also select variables for viewing by specifying one or more MOOS processes with the command line option `--src=<process>,<process>,...`. All variables from these processes will then be included in the scope list.

In the *AllVars* sub-mode, all MOOS variables in the MOOSDB are displayed, unless explicitly filtered out. The most common way of filtering out variables in the *AllVars* sub-mode is to provide a filter string interactively by typing the '/' key and entering a filter. Only lines that contain this string as a substring in the variable name will then be shown. The filter may also be provided on startup with the `--filter=pattern` command line option.

In both sub-modes, variables that would otherwise be included in the report may be masked out with two further options. Variables that have never been written to by any MOOS process are referred to as virgin variables, and by default are shown with the string "n/a" in their value column. These may be shut off from the command line with `--mask=virgin`, or in the MOOS configuration block by including the line `display_virgins=false`. Similarly, variables with an empty string value may be masked out from the command line with `--mask=empty`, or with the line `display_empty_strings=false` in the MOOS configuration block of the `.moos` file.

5.3.2 The History Content Mode

In the *history* content mode, the recent values for a single MOOS variable are reported. Contrast this with the *scoping* mode where a snapshot of a variable value is displayed, and that value may have changed several times between successive reports. The output generated in this mode may look like that in Figure 36 which shows the desired heading of a vehicle going into the first turn of the alpha mission. This **uXMS** session can be launched from the command line with:

```
$ uXMS alpha.moos --history=DESIRED_HEADING
```

VarName	(S)ource	(T)ime	VarValue (HISTORY:EVENTS)
DESIRED_HEADING	pHelmiVp	21.67	(1) 113
DESIRED_HEADING	pHelmiVp	24.94	(13) 114
DESIRED_HEADING	pHelmiVp	27.45	(10) 113
DESIRED_HEADING	pHelmiVp	29.72	(9) 114
DESIRED_HEADING	pHelmiVp	31.98	(9) 113
DESIRED_HEADING	pHelmiVp	34.24	(9) 114
DESIRED_HEADING	pHelmiVp	36.51	(9) 113
DESIRED_HEADING	pHelmiVp	36.76	(1) 157
DESIRED_HEADING	pHelmiVp	37.01	(1) 158
DESIRED_HEADING	pHelmiVp	37.26	(1) 160
DESIRED_HEADING	pHelmiVp	37.51	(1) 162
DESIRED_HEADING	pHelmiVp	37.76	(1) 163
DESIRED_HEADING	pHelmiVp	38.01	(1) 165
DESIRED_HEADING	pHelmiVp	38.26	(1) 166
DESIRED_HEADING	pHelmiVp	38.51	(1) 167
DESIRED_HEADING	pHelmiVp	38.77	(1) 169
DESIRED_HEADING	pHelmiVp	39.02	(1) 171
DESIRED_HEADING	pHelmiVp	39.27	(1) 172
DESIRED_HEADING	pHelmiVp	39.52	(1) 174
DESIRED_HEADING	pHelmiVp	39.77	(1) 177
DESIRED_HEADING	pHelmiVp	40.02	(1) 178
DESIRED_HEADING	pHelmiVp	40.53	(2) 179
DESIRED_HEADING	pHelmiVp	41.28	(3) 180
DESIRED_HEADING	pHelmiVp	42.03	(3) 181
DESIRED_HEADING	pHelmiVp	42.53	(2) 182
DESIRED_HEADING	pHelmiVp	43.04	(2) 183
DESIRED_HEADING	pHelmiVp	43.79	(3) 184
DESIRED_HEADING	pHelmiVp	44.55	(3) 185
DESIRED_HEADING	pHelmiVp	45.80	(5) 186
DESIRED_HEADING	pHelmiVp	46.81	(4) 185
DESIRED_HEADING	pHelmiVp	47.81	(4) 184
DESIRED_HEADING	pHelmiVp	48.82	(4) 183
DESIRED_HEADING	pHelmiVp	50.58	(7) 182
DESIRED_HEADING	pHelmiVp	55.10	(18) 181
DESIRED_HEADING	pHelmiVp	69.42	(57) 180

Figure 36: **A uXMS scope on a single variable history:** A vehicle's desired heading is monitored as it goes into the first turn of the alpha mission. Values in parentheses indicate the number of successive postings without a change of value.

The output structure in the *history* mode is the same as in the *scoping* mode in terms of what data is in the columns and header lines. Each line however is dedicated to the same variable and shows the progression of values through time. To save screen real estate, successive mail received for with identical source and value will consolidated on one line, and the number in parentheses is merely incremented for each such identical mail. For example, the last line shown in Figure 36, the value of **DESIRED_HEADING** has remained the same for 57 consecutives posts to the MOOSDB.

The output in the *history* mode may be adjusted in a few ways:

- *Modifying the number of history lines:* The number of lines of history may be increased or decreased by hitting the 'greater than' or 'less than' keys respectively. A maximum of 100 and minimum of 5 lines is allowed. The default is 40.
- *Setting the history variable:* The history variable may be set on the command line with `--history=VAR`, or set in the mission file with `history_var=<MOOSVar>`. If set in both, the command line setting takes precedent.
- *Hiding the history variable:* To increase the available real estate on each line, the variable name column may be suppressed or restored by toggling the 'j' key. The history variable is shown by default but may be configured to be off upon startup by setting `display_history_var=false` in the mission file.

Presently there is no way to dynamically change the history variable, or scope on more than one variable's history. (But you can open more than one `uXMS` session to scope on more than one variable's history.)

5.3.3 The Processes Content Mode

In the *processes* content mode running processes may be monitored and selected for either including or excluding variables from the selected processes. This mode may be toggled with the 'p' key. For example, launching the alpha mission and then `uXMS` from the command line:

```
$ uXMS alpha.moos DESIRED_HEADING
```

After `uXMS` is launched, toggle into the processes content mode with the 'p'. This should present something similar to Figure 37.

ID	Process Name	Mail	Client
1	pHelmIpvP	67.61	1.61
a	pNodeReporter	-	1.61
b	uProcessWatch	-	1.61
c	pMarineViewer	-	1.61
d	pMarinePID	-	1.61
e	uSimMarine	0.00	1.61
f	pLogger	-	1.61

Figure 37: **A `uXMS` processes content mode:** All processes known to the scope (via the `DB_CLIENTS` variable) are shown. The Mail column shows the time since mail has been received from the client. The Client column shows the time since the client has shown up on the `DB_CLIENTS` list.

The first two columns show the processes known to `uXMS` and a process ID randomly assigned to each process. These IDs may be used select a process as explained shortly. The last two columns

show the time since mail was last received and the time since the process last appeared on the `DB_CLIENTS` list. Try killing one of the processes and see what happens.

By default `uXMS` tries to make use of information produced by `uProcessWatch`. The first line in the body of the report in Figure 37 shows the contents of the `PROC_WATCH_SUMMARY` variable. In this example, mail has only been received from `pHelmIvP` and `uProcessWatch`. The former because `uXMS` was launched from the command line scoping on `DESIRED_HEADING`, and the latter because `uXMS` is automatically configured to receive the `PROC_WATCH_SUMMARY` mail from `uProcessWatch`. If `uProcessWatch` is not running the report will simply state so.

Perhaps the most useful feature of the `processes` content mode is the ability to select a process to either include or exclude variables published by that process on the watch list. To *include* variables from a process, type the '+' key, and a menu and prompt like that shown in Figure 38

ID	Process Name	Mail	Client
1	pHelmIvP	6.04	1.61
a	pNodeReporter	-	1.61
b	pMarineViewer	-	1.61
c	pMarinePID	-	1.61
d	uSimMarine	0.00	1.61
e	pLogger	-	1.61

Include a process > []

Figure 38: Adding watch-list variables based process inclusion: All processes known to the scope (via the `DB_CLIENTS` variable) are shown. Each process has a single-character ID which may be entered at the prompt to select the process for inclusion.

Once a process has been selected, `uXMS` will subscribe for all variables published by the selected process. Note this is different from subscribing for mail solely produced by the selected MOOS app since the same variable(s) may be also published by other MOOS applications. The example in Figure 38 is also from the alpha mission. If the `pMarineViewer` application were selected, a scoping report something like that below in Figure 39 would result.

```

uXMS_995 alpha
=====
0/0(1156)

VarName      (S)ource      (T)      (C)      VarValue (SCOPING:EVENTS)
-----
APPCAST          uSimMarine      "proc=uSimMarine"
APPCAST_REQ        n/a           n/a
APPCAST_REQ_ALL    pMarineViewer   "node=all,app=all,duration=3.0,key..."
APPCAST_REQ_ALPHA  pMarineViewer   "node=alpha,app=uSimMarine,duratio..."
DESIRED_HEADING   pHelmIvp       0
HELM_MAP_CLEAR     pMarineViewer   0
NAV_X              uSimMarine      0
PMV_CONNECT        pMarineViewer   0

```

Figure 39: **Augmented scoping report:** The variables published by `pMarineViewer` are now included in the scoping report after this process was selected for inclusion.

Once a process has been added or excluded, its status will be indicated next time the processes mode is entered, with either a '+' or '-' next to the process ID. For example, the report shown in Figure 40 below is generated after hitting the '-' key to select a process for exclusion. The plus sign next to the `pMarineViewer` indicates that it has been selected for inclusion previously.

ID	Process Name	Mail	Client
1	pHelmIvp	14.08	1.61
a	pNodeReporter	-	1.61
b	uProcessWatch	-	1.61
+c	pMarineViewer	0.00	1.61
d	pMarineID	-	1.61
e	uSimMarine	0.00	1.61
f	pLogger	-	1.61

(PROCESSES:PAUSED)

Exclude a process > []

Figure 40: **Excluding watch-list variables based on process origin:** The process list includes an indicator to the left of the ID showing whether the process is presently included ('+') or excluded ('-'). In this case the `pMarineViewer` application has been included but no action has been taken regarding any other processes.

When a process or application has been selected for *exclusion*, this is handled in the following way. When a scoping report is being generated, if a particular variable has most recently been set by an excluded process, it is not include in the scoping report. Note, it may have also been published by another application not on the exclusion list.

5.4 Configuration File Parameters for uXMS

Configuratton of `uXMS` may be done from a configuration file (`.moos` file), from the command line, or both. Generally the parameter settings given on the command line override the settings from the `.moos` file, but using the configuration file is a convenient way of ensuring certain settings are in effect on repeated command line invocations. The following is short description of the parameters:

Listing 5.1: Configuration Parameters for uXMS.

```
colormap: Associates a color for the line of text reporting the given variable.  
content_mode: Set content mode to either scoping, history, procs, or help.  
display_all: If true, all variables are reported in the scoping content mode.  
display_aux_source: If true, non-null auxilliary source is shown in place of source.  
display_community: If true, the Community column is rendered.  
display_history_var: If false, history var not shown in history mode.  
display_source: If true, the Source column is rendered.  
display_time: If true, the Time column is rendered.  
display_virgins: If false, variables never written to the MOOSDB are not reported.  
history_var: Names the MOOS variable reported in the history mode.  
refresh_mode: Determines when new reports are written to the screen.  
source: Names a MOOS app for which all variables will be scoped.  
term_report_interval: Time (secs) between report updates (default 0.6).  
trunc_data: If true, variable string values are truncated.  
var: A comma-separated list of variables to scope on in the scoping mode.
```

An Example uXMS Configuration Block

An example configuration is given in Listing 2. This may also be elicited from the command line:

```
$ uXMS --example or -e
```

Listing 5.2: An example uXMS configuration block.

```
1 ProcessConfig = uXMS  
2 {  
3     AppTick    = 4  
4     CommsTick = 4  
5  
6     var      = NAV_X, NAV_Y, NAV_SPEED, NAV_HEADING  
7     var      = PROC_WATCH_SUMMARY  
8     var      = PROC_WATCH_EVENT  
9     source   = pHelmIpv, pMarineViewer  
10  
11    history_var        = DB_CLIENTS  
12  
13    display_virgins    = true    // default  
14    display_source      = false   // default  
15    display_aux_source  = false   // default  
16    display_time        = false   // default
```

```

17  display_community    = false   // default
18  display_all          = false   // default
19  trunc_data           = 40     // default is no truncation.
20
21  term_report_interval = 0.6    // default (seconds)
22
23  color_map      = pHelmIvP, red // All postings by pHelmIvP red
24  color_map      = NAV_SPEED, blue // Only var NAV_SPEED is blue
25
26  refresh_mode = events      // default (or streaming/paused)
27  content_mode = scoping     // default (or history,procs)
28 }

```

5.4.1 The colormap Configuration Parameter

Most of the the configurable options deal with content and layout of the information in the terminal window, but color can also be used to facilitate monitoring one or more variables. The parameter

```
colormap = <variable/app>, <color>
```

is used to request that a line in the report containing the given variable or produced by the given MOOS application (source) is rendered in the given color. The choices for color are limited to red, green, blue, cyan, and magenta.

5.4.2 The content_mode Configuration Parameter

The content mode determines what information is generated in each report to the terminal output (Section 5.3). This mode is set with the following parameter:

```
content_mode = <mode-type> // Default is "scoping"
```

The default setting is "scoping" to select the scoping content mode described in Section 5.3.1. It may also be set to "history" to select the history mode described in Section 5.3.2, or set to "procs" to select the processes mode described in Section 5.3.3.

5.4.3 The display* Configuration Parameters

In the scoping and history content modes, the uXMS report has columns of data that may be optionally turned off to conserve real estate, the *Time*, *Source* and *Community* columns as shown in Figures 33, 36, and 39. By default they are turned off, and they may be toggled on and off by the user at run time. Their initial state may also be configured with the following three parameters:

```

display_community  = <Boolean> // Default is false
display_source    = <Boolean> // Default is false
display_time      = <Boolean> // Default is false
display_aux_source = <Boolean> // Default is false

```

The `display_aux_source` parameter, when true, not only activates this column, but also indicates that the auxilliary source is to be shown instead of the source. Not all MOOS variable postings have the auxilliary source field filled in. In the case of variables posted by the helm, however, this field contains the both the helm iteration and name of the behavior. If the auxilliary source is empty for a particular variable, the primary source is shown instead. To be clear what is being shown, the auxilliary source is always contained in brackets. For example, [241:waypoint_return], may indicate the variable was posted by the helm on iteration 241 by the waypoint return behavior.

The `display_all` parameter determines whether the scope list contains only those variables specified by the user, or all MOOS variables published by any MOOS process. The latter is useful at times when you can't quite remember the variable name you're looking for or who publishes it. When displaying all variables, certain variables may be masked out by selecting a process (MOOS app) to exclude, as described in Section 5.3.3. It may also be enabled with the 'A' key, and disabled with the 'a' key at the terminal at run time. It may also be enabled from the command line with the `--all` or `-a` switches.

```
display_all = <Boolean> // Default is false
```

Using the `display_virgins` parameter, the report content may be further modified to mask out lines containing variables that have never been written to, and variables with an empty-string value. This is done with the below configuration line. It may also be toggled with the 'v' key at the terminal at run time, and it may also be specified from the command line with the `--novirgins` or `-g` switches.

```
display_virgins = <Boolean> // Default is true
```

In the history mode, the name of the variable is the same on each line. This makes it clear what variable is being shown, but takes up screen real estate and is redundant. It may be suppressed by setting `display_history_var` to false in the mission file. It may also be toggled with the 'j' key at the terminal at run time.

```
display_history_var = <Boolean> // Default is true
```

5.4.4 The `history_var` Configuration Parameter

The variable reported in the *history* mode is set with the below configuration line:

```
history_var = <MOOSVar>
```

The history report only allows for one variable, and multiple instances of the above line will simply honor the last line provided. The history variable is also automatically added to the watch list used in the scoping mode. The history variable may also be set on the command line when `uXMS` is launched from the terminal, with `--history=<MOOS-variable>`. If set in both places, the command line choice overrides the choice in the mission file.

5.4.5 The `refresh_mode` Configuration Parameter

The *refresh* mode determines when new reports are generated to the screen, as discussed in Section 5.2. It is set with the below configuration line:

```
refresh_mode = <mode> // Valid modes are "paused", "streaming", "events"
```

The initial refresh mode is set to "events" by default. The refresh mode set in the configuration file may be overridden from the command line with `--mode=paused|events|streaming`, or chosen interactively at run time with the '`e`' key for *events*, the spacebar key for *paused*, or the '`r`' key for *streaming*.

5.4.6 The `source` Configuration Parameter

The variable scope list may be set or augmented by naming a particular MOOS app source with the below parameter:

```
source = <MOOSApp>, <MOOSApp>, ...
```

With this, `uXMS` will subscribe for any MOOS variable published by the named application(s). Since variables may be published by multiple applications, don't be surprised to see postings made by other applications. This is *not* a request to receive mail only from the named source(s). Sources may also be chosen from the command line with the `--src=<MOOSApp>,<MOOSApp>,...K` command line switch. Sources may also be included or excluded dynamically in the `processes` content mode as described in Section 5.3.3, with the '+' and '-' keys.

5.4.7 The `term_report_interval` Configuration Parameter

The `term_report_interval` is a parameter defined for all AppCasting MOOS applications. It specifies the amount of time between success updates to the terminal. Report updates are not based on the application's apptick since this may be considerably faster than the human may absorb and wastes CPU resources. The default refresh rate is 0.6 seconds between refreshes. This may be overridden with:

```
term_report_interval = <Non-Zero Value> // Default is 0.6 seconds
```

The interval may also be specified on the command line with the `--termint=<Non-Zero Value>` switch. The accepted range, in seconds, is [0, 10]. Keep in mind that report frequency cannot be any faster than the actual apptick set for `uXMS`.

5.4.8 The `trunc_data` Configuration Parameter

The current value of a MOOS variable is shown in the `VarValue` column in both the scoping and history content modes. This value may be quite long and overwrap several lines and make things hard to read. The user can choose to truncate the content by setting `trunc_data` parameter:

```
trunc_data = <unsigned int> // Default is zero (no truncating)
```

Values are accepted in the range [10, 1000]. Truncated string output will be further indicated by adding a trailing "..." to the end of the output. Truncation may be toggled on/off at run time by hitting the `` (back-tick) key. The default truncation length is 40 characters. The length of truncated output may also be adjusted at run time with the '{' and '}' keys.

5.4.9 The `var` Configuration Parameter

The variables reported on in the *scoping* mode, the scope list, are declared with configuration lines of the form:

```
var = <MOOSVar>, <MOOSVar>, ...
```

Multiple such lines, each perhaps with multiple variables, are accommodated. The scope list may be *augmented* on the command line by simply naming variables as command line arguments. The scope list provided on the command line may *replace* the list given in the configuration file if the --clean command line option is also invoked.

5.5 Command Line Usage of uXMS

Many of the parameters available for setting the .moos file configuration block can also be affected from the command line. The command line configurations always trump any configurations in the .moos file. As with the `uPokeDB` application, the server host and server port information can be specified from the command line too to make it easy to open a `uXMS` window from anywhere within the directory tree without needing to know where the .moos file resides. A `uXMS` session can be launched to connect to the MOOSDB of a remote vehicle on the network, if the IP address and port number are known, with:

```
$ uXMS --server_host=10.25.0.191 --server_port=9000 --src=pHelmIvP
```

The basic command line usage for the `uXMS` application is the following:

```
$ uXMS --help or -h
```

Listing 5.3: Command line usage for the `uXMS` tool.

```
1 Usage: uXMS [file.moos] [OPTIONS]
2 Options:
3   --alias=<ProcessName>
4       Launch uXMS with the given process name rather than uXMS.
5   --all,-a
6       Show ALL MOOS variables in the MOOSDB
7   --clean,-c
8       Ignore scope variables in file.moos
```

```

9  --colormap=<MOOSVar>,<color>
10     Display all entries where the variable, source, or community
11     has VAR as substring. Allowable colors: blue, red, magenta,
12     cyan, or green.
13  --colorany=<MOOSVar>,<MOOSVar>,...
14     Display all entries where the variable, community, or source
15     has VAR as substring. Color auto-chosen from unused colors.
16  --example, -e
17      Display example MOOS configuration block.
18  --help,-h
19      Display this help message.
20  --history=<MOOSVar>
21      Allow history-scoping on variable
22  --interface,-i
23      Display MOOS publications and subscriptions.
24  --novirgins,-g
25      Don't display virgin variables
26  --mode=[paused,EVENTS,streaming]
27      Determine display mode. Paused: scope updated only on user
28      request. Events: data updated only on change to a scoped
29      variable. Streaming: updates continuously on each app-tick.
30  --server_host=<IPAddress>
31      Connect to MOOSDB at IP=value, not from the .moos file.
32  --server_port=<PortNumber>
33      Connect to MOOSDB at port=value, not from the .moos file.
34  --show=[source,time,community,aux]
35      Turn on data display in the named column, source, time, or
36      community. All off by default enabling aux shows the
37      auxilliary source in the souce column.
38  --src=<MOOSApp>,<MOOSApp>, ...
39      Scope only on vars posted by the given MOOS processes
40  --trunc=value [10,1000]
41      Truncate the output in the data column.
42  --termint=value [0,10] (default is 0.6)
43      Minimum real-time seconds between terminal reports.
44  --version,-v
45      Display the release version of uXMS.
46
47 Shortcuts
48
49  -t Short for --trunc=25
50  -p Short for --mode=paused
51  -s Short for --show=source
52  -st Short for --show=source,time

```

Using the `--clean` switch will cause `uXMS` to ignore the variables or sources specified in the `.moos` file configuration block and only scope on the variables specified on the command line (otherwise the union of the two sets of variables is used). Typically this is done when a user wants to quickly scope on a couple variables and doesn't want to be distracted with the longer list specified in the `.moos` file. Arguments on the command line other than the ones described above are treated as variable requests.

If the `server_host` or the `server_port` are not provided on the command line, and a MOOS file is also not provided, the user will be prompted for the two values. Since the most common scenario

is when the MOOSDB is running on the local machine ("localhost") with port 9000, these are the default values and the user can simply hit the return key.

```
$ uXMS
$ Enter Server: [localhost] <return>
    The server is set to "localhost"
$ Enter Port: [9000] <return>
$     The port is set to "9000"
```

5.6 Console Interaction with uXMS at Run Time

Many of the launch-time configuration parameters may be altered at run time through interaction with the console window. For example, the displaying of the Source column is configured to be false by default, may be configured in the mission file with `display_source=true`, and may be configured on the command line with `--show=source`. It may also be toggled at run time by typing the 's' character at the console window. A list of run-time key mappings may be shown at any time by typing the 'h' key for help. Re-hitting this key resumes prior scoping. Listing 4 shows the contents of the help menu.

Listing 5.4: The help-menu on the uXMS console.

1	KeyStroke	Function	(HELP)
2	-----	-----	-----
3	s	Toggle show source of variables	
4	t	Toggle show time of variables	
5	c	Toggle show community of variables	
6	v	Toggle show virgin variables	
7	x	Toggle show Auxilliary Src if non-empty	
8	d	Content Mode: Scoping Normal	
9	h	Content Mode: Help. Hit 'R' to resume	
10	p	Content Mode: Processes Info	
11	z	Content Mode: Variable History	
12	> or <	Show More or Less Variable History	
13	} or {	Show More or Less Truncated VarValue	
14	/	Begin entering a filter string	
15	'	Toggle Data Field truncation	
16	?	Clear current filter	
17	a	Revert to variables shown at startup	
18	A	Display all variables in the database	
19	u/SPC	Refresh Mode: Update then Pause	
20	r	Refresh Mode: Streaming	
21	e	Refresh Mode: Event-driven refresh	

5.7 Running uXMS Locally or Remotely

The choice of `uXMS` as a scoping tool was designed in part to support situations where the target MOOSDB is running on a vehicle with low bandwidth communications, such as an AUV sitting on the surface with only a weak RF link back to the ship. There are two distinct ways one can run `uXMS` in this situation and its worth noting the difference. One way is to run `uXMS` locally on one's own machine, and connect remotely to the MOOSDB on the vehicle. The other way is to log onto

the vehicle through a terminal, run `uXMS` remotely, but in effect connecting locally to the MOOSDB also running on the vehicle.

The difference is seen when considering that `uXMS` is running three separate threads. One accepts mail delivered by the MOOSDB, one executes the iterate loop of `uXMS` where reports are written to the terminal, and one monitors the keyboard for user input. If running `uXMS` locally, connected remotely, even though the user may be in paused mode with no keyboard interaction or reports written to the terminal, the first thread still may have a communication requirement perhaps larger than the bandwidth will support. If running remotely, connected locally, the first thread is easily supported since the mail is communicated locally. Bandwidth is consumed in the second two threads, but the user controls this by being in paused mode and requesting new reports judiciously.

5.8 Connecting multiple `uXMS` processes to a single MOOSDB

Multiple versions of `uXMS` may be connected to a single MOOSDB. This is to simultaneously allow several people a scope onto a vehicle. Although MOOS disallows two processes of the same name to connect to MOOSDB, `uXMS` generates a random number between 0-999 and adds it as a suffix to the `uXMS` name when connected. Thus it may show up as something like `uXMS_871` if you scope on the variable `DB_CLIENTS`. In the unlikely event of a name collision, the user can just try again.

5.9 Using `uXMS` with Appcasting

Appcasting allows a really useful way of using `uXMS`, especially in the case of multiple deployed vehicles. Prior to appcasting, the only way to use `uXMS` is through a terminal window. With appcasting the `uXMS` report may also be published to the MOOSDB for remote viewing via a `uMAC` utility or with `pMarineViewer`.

For example, consider a case where some number of vehicles are deployed, each with an interface to their batteries, compass and GPS. The interfaces may be named `iBatteryMonitor`, `iCompass`, and `iGPS`. Each interface publishes several MOOS variables including health status messages. A mission could be configured with three `uXMS` processes launched at mission startup with something similar to:

Listing 5.5: Launching several `uXMS` processes with appcasting.

```
1 ProcessConfig = ANTLER
2 {
3     MSBetweenLaunches = 200
4
5     Run = MOOSDB          @ NewConsole = false
6     // Other MOOS Apps
7     Run = iGPS            @ NewConsole = false
8     Run = iBatteryMonitor @ NewConsole = false
9     Run = iCompass         @ NewConsole = false
10    Run = uXMS            @ NewConsole = false ~ uXMS_GPS
11    Run = uXMS            @ NewConsole = false ~ uXMS_BATTERY_MONITOR
12    Run = uXMS            @ NewConsole = false ~ uXMS_COMPASS
13 }
14
15 ProcessConfig = uXMS_GPS
16 {
```

```

17     SOURCE = iGPS
18 }
19 ProcessConfig = uXMS_BATTERY_MONITOR
20 {
21     SOURCE = iBatteryMonitor
22 }
23 ProcessConfig = uXMS_COMPASS
24 {
25     SOURCE = iCompass
26 }

```

With this configuration the three **uXMS** processes will launch, each *without* a terminal window open, and each with a different descriptive name. The **uXMS** reports are accessible with any of the appcast viewing tools, **uMAC**, **uMACView**, and **pMarineViewer**. In the latter two tools for example, the **uXMS** reports may appear in a menu selection like that shown in Figure 41.

Node	AC	CW	RW	App	AC	CW	RW
shoreside	74	0	0	uFldNodeBroker	7	0	0
henry	235	0	0	uSimMarine	3	0	0
gilda	34	0	0	pNodeReporter	3	0	0
				uXMS_BATTERY_MONITOR	3	0	0
				pHelmIpv	3	0	0
				uFldMessageHandler	3	0	0
				pBasicContactMgr	3	0	0
				uXMS_COMPASS	8	0	0
				uXMS_GPS	196	0	0
				pHostInfo	3	0	0
				uProcessWatch	3	0	0

uXMS_GPS henry				0/0(673)
VarName	(S)	(T)ime	(C)	VarValue (SCOPING:EVENTS)
GPS_HEADING	40.14			"18.3,"
GPS_LATITUDE	40.14			"43.825304,"
GPS_LONGITUDE	40.14			"-70.330402,"
GPS_MAGNETIC_DECLINATION	40.14			"12.3,"
GPS_SPEED	40.14			0
GPS_X	40.14			"19.3,"
GPS_Y	40.14			923.1
GPS_YAW	40.14			"1.09,"

Figure 41: **Appcasting and uXMS:** Multiple vehicles are each configured with three dedicated **uXMS** processes to scope on variables particular to a given device or sensor. The **uMAC** viewer interface allows the user to select any vehicle and select a **uXMS** report to see the desired information for that vehicle and device.

In this way the user may monitor the health of these three instruments across all fielded vehicles with a single GUI without having to write any special code for these devices.

5.10 Publications and Subscriptions for uXMS

The interface for **uXMS**, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uXMS --interface or -i
```

5.10.1 Variables Published by uXMS

- **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 5.9.

5.10.2 Variables Subscribed for by uXMS

- **APPCAST_REQ**: A request to generate and post a new appcast report, with reporting criteria, and expiration. Section 5.9.
- **DB_CLIENTS**: To handle requests to scope on all variables.
- **DB_UPTIME**: To determine the MOOSDB start time. All **uXMS** times reported are times since MOOSDB started.
- **PROC_WATCH_SUMMARY**: As a convenience this summary is displayed in the *processes* content mode. It is posted by **uProcessWatch**.
- **USER-DEFINED**: The variables subscribed for are those on the *scope list*, augmented with the **var** and **source** parameters described in Sections 5.4.6 and 5.4.9.

6 uTimerScript: Scripting Events to the MOOSDB

6.1 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 6.1.

Listing 6.1: A Simple Timer Script.

```
1 ProcessConfig = uTimerScript
2 {
3     event = var=MOOS_MANUAL_OVERRIDE, val=false
4     event = var=DEPLOY, val=true
5     event = quit
6 }
```

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.

6.2 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.

6.2.1 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 6.4.

6.2.2 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 6.2.3.

6.2.3 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 6.3.1. 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 6.3.1. See the example in Section 6.9.1 for a case where the `upon_awake` feature is handy.

6.3 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.

6.3.1 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`.

6.3.2 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 6.3.1, 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 6.9.1 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 unpauseable with respect to the logic conditions. This is configured with:

```
script_atomic = <Boolean>
```

It can however be paused and unpauseable via the pause variable, e.g., `UTS_PAUSE`, as described in Section 6.3.1. 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.

6.3.3 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`.

6.3.4 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.

6.4 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.

6.4.1 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.

6.4.2 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.

6.4.3 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.

6.5 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.

6.5.1 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 6.9.2 for a use of random time warping to simulate random wind gusts.

6.5.2 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 6.9.1 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.

6.5.3 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 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 6.3.1) and `conditions_ok=true` (Section 6.3.2). It has been reset twice out of a maximum of five allowed resets (`resets=2/5`, Section 6.2.3). Time warping is being deployed (`time_warp=3`, Section 6.5), there is no start delay in use (`start_delay=0`, Section 6.5.2). The shuffle feature is turned off (`shuffle=false`, Section 6.2.3). The script is not configured to reset upon re-entering the un-paused state (`awake_reset=false`, Section 6.2.3).

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"
```

6.6 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 2. 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 6.5.1). Lines 8-9 show the delay applied at the start and after a script reset (Section 6.5.2). Line 10 indicates the script is presently not paused (Section 6.3.1). Line 11 indicates the script presently meets any prevailing logic conditions (Section 6.3.2). 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 6.4.2.

Listing 6.2: Example uTimerScript console and appcast output.

```
1 =====
2 uTimerScript charlie                      0/0(450)
3 =====
4 Current Script Information:
5   Elements: 10(8)
6   Reinit: 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.

6.7 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 6.3: Configuration Parameters for uTimerScript.

- condition:** A logic condition that must be met for the script to be un-paused. Section [6.3.2](#).
- 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 [6.5.2](#).

<code>delay_start</code> :	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 6.5.2 .
<code>event</code> :	A description of a single event in the timer script. Section 6.2.1 .
<code>forward_var</code> :	A MOOS variable for taking cues to forward time. The default is <code>UTS_FORWARD</code>). Section 6.3.3 .
<code>paused</code> :	A Boolean indicating whether the script is paused upon launch. Legal values: true, false. The default is false. Section 6.3.1 .
<code>pause_var</code> :	A MOOS variable for receiving pause state cues (<code>UTS_PAUSE</code>). Section 6.3.1 .
<code>rand_var</code> :	A declaration of a random variable macro to be expanded in event values. Section 6.4.2 .
<code>reset_max</code> :	The maximum amount of resets allowed. Legal values: any non-negative integer, or the string "nolimit". The default is "nolimit". Section 6.2.3 .
<code>reset_time</code> :	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 6.2.3 .
<code>reset_var</code> :	A MOOS variable for receiving reset cues. The default is <code>UTS_RESET</code> .
<code>script_atomic</code> :	When <code>true</code> , a started script will complete if conditions suddenly fail. Legal values: true, false. The default is false.
<code>script_name</code> :	Unique (hopefully) name given to this script. The default is "unnamed".
<code>shuffle</code> :	If <code>true</code> , timestamps are recalculated on each reset of the script. Legal values: true, false. The default is <code>true</code> . Section 6.2.3 .
<code>status_var</code> :	A MOOS variable for posting status summary. The default is <code>UTS_STATUS</code> . Section 6.5.3
<code>time_warp</code> :	Rate at which time is accelerated in executing the script. Legal values: any non-negative number. The default is zero. Section 6.5 .
<code>upon_awake</code> :	Reset or re-start the script upon conditions being met after failure ("n/a"). Section 6.2.3 .
<code>verbose</code> :	If <code>true</code> , progress output is generated to the console (<code>true</code>).

6.8 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 or -i
```

6.8.1 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 6.6.
- **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 6.5.3.

6.8.2 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 appcast report, with reporting criteria, and expiration. Section 11.10.4.
- **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 `uTimerScript` will quit. See Section 6.3.4.
- **UTS_NEXT**: When received with the value "next", the script will fast-forward in time to the next event. See Section 6.3.3.
- **UTS_RESET**: When received with the value of either "true" or "reset", the timer script will be reset. See Section 6.2.3.
- **UTS_FORWARD**: When received with a numerical value greater than zero, the script will fast-forward by the indicated time. See Section 6.3.3.
- **UTS_PAUSE**: When received with the value of "true", "false", "toggle", the script will change its pause state correspondingly. See Section 6.3.1.

In addition to the above MOOS variables, `uTimerScript` will subscribe for any variables involved in logic conditions, described in Section 6.3.2.

6.8.3 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 4 below.

Listing 6.4: 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 unpause
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 }

```

6.9 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.

6.9.1 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 42, 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.

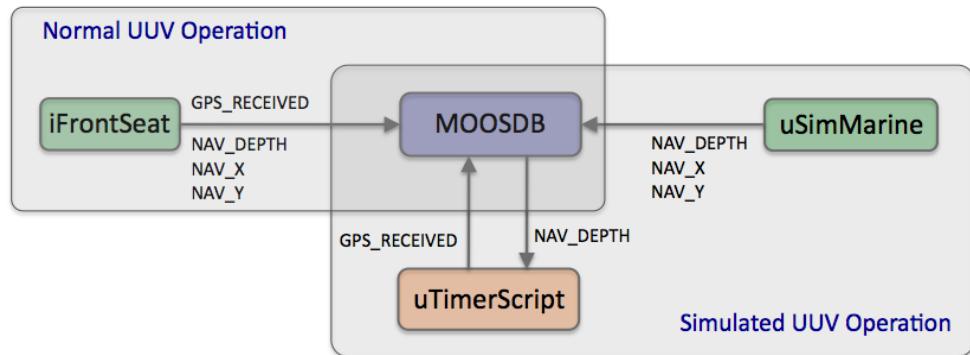


Figure 42: **Simulating a GPS Acknowledgment:** In a physical operation of the vehicle, the navigation solution and a `GPS_UPDATE_RECEIVED` heartbeat are received from the main vehicle (front-seat) computer via a MOOS module acting as an interface to the front-seat computer. In simulation, the navigation 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 5 gets the job done.

Listing 6.5: 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 6.4.1 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.

6.9.2 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 43), 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`.

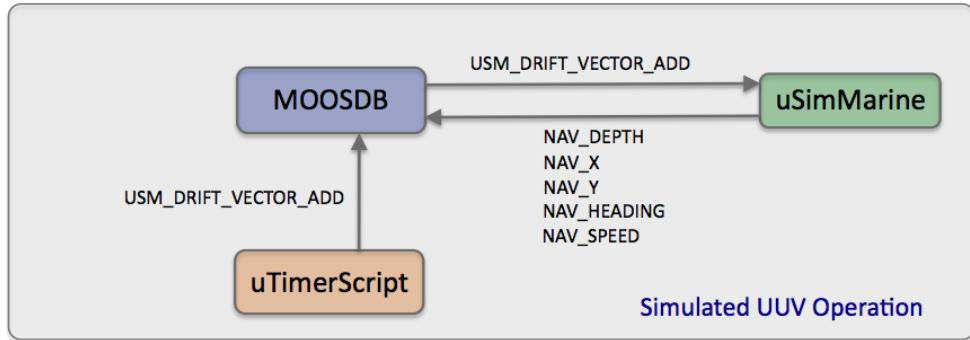


Figure 43: **Simulated Wind Gusts:** The `uTimerScript` application may be configured to post periodic sequences of external drift values, used by the `uSimMarine` application to simulate wind gust effects on its simulated vehicle.

The script in Listing 6 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 6.6: A `uTimerScript` configuration for simulating simple wind gusts.

```

1 //-----
2 // uTimerScript configuration block
3
4 ProcessConfig = uTimerScript
5 {
6     AppTick      = 2
7     CommsTick   = 2
8
9     paused       = false
10    reset_max    = unlimited
11    reset_time   = end
12    delay_reset  = 10:60
13    time_warp    = 0.25:2.0
14    script_name   = WIND
15    script_atomic = true
16
17    randvar = varname=ANG, min=0,    max=359, key=at_reset
18    randvar = varname=MAG, min=0.5,  max=1.5, key=at_reset
19
20    event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*0.2}", time=0
21    event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*0.2}", time=2
22    event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*0.2}", time=4
23    event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*0.2}", time=6
24    event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*0.2}", time=8
25
26    event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*-0.2}", time=10
27    event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*-0.2}", time=12
28    event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*-0.2}", time=14
29    event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*-0.2}", time=16

```

```
30     event = var=USM_DRIFT_VECTOR_ADD, val="$[ANG],{$[MAG]*-0.2}", time=18
31 }
```

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 [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.

7 pBasicContactMgr: Managing Platform Contacts

7.1 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.

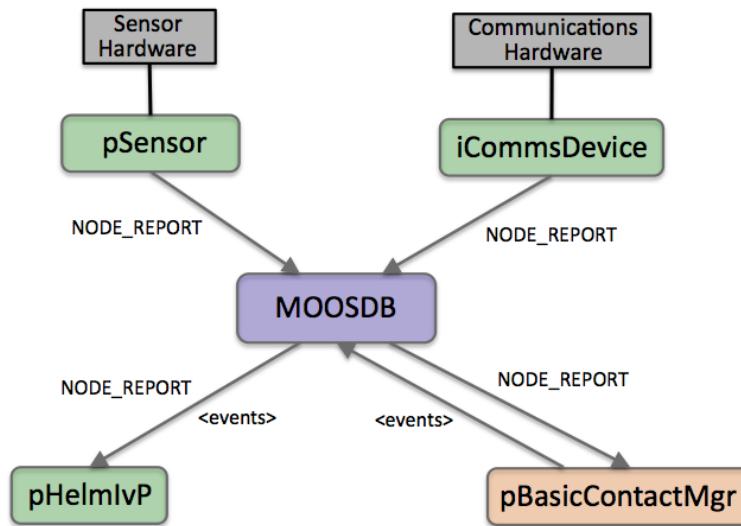


Figure 44: **The pBasicContactMgr Application:** The `pBasicContactMgr` utility receives `NODE_REPORT` information from other MOOS applications and manages a list of unique contact records. It may post additional user-configurable alerts to the MOOSDB based on the contact information and user-configurable 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.

7.2 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.

7.2.1 Contact Alert Configuration

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 5.

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 B 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 an 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 5 for an example alert configuration.

7.2.2 Dynamic Contact Alert Configuration

Alert configuration may also be handled dynamically via an incoming MOOS variable, `BCM_ALERT_REQUEST`. This has the same format used in the `alert` configuration lines described above. For example, the `pBasicContactMgr` configuration lines:

Listing 7.1: Configuration snippet for `pBasicContactMgr`.

```

1 //-----
2 // pBasicContactMgr Configuration Block3
4
5 ProcessConfig = pBasicContactMgr
6 {
7     . . .
8     alert = id=avd, var=CONTACT_INFO, val="name=$[VNAME] # contact=$[VNAME]"
9     alert = id=avd, alert_range=80, alert_range_color=green
10    . . .
11 }
```

could be achieved with the following posting to the MOOSDB:

```
BCM_ALERT_REQUEST = id=avd, var=CONTACT_INFO, val="name=$[VNAME] # contact=$[VNAME]",
                    alert_range=80, alert_range_color=green
```

Who makes this posting and why is this preferable? The answer to the first question is the client, or the intended consumer of the alert. The reason why this is preferable is a bit more subtle. The short answer is that the client knows best. Limiting the alert configuration to the client can simplify mission configuration and reduce unintended mis-configurations.

Consider the case, for example, with the CollisionAvoidance behavior. This behavior is typically configured to complete, and die, when the behavior contact exceeds a certain range. This range is set by the user in the behavior parameter `completed_dist`. A situation to avoid would be the case where the behavior completes but then is immediately spawned again due to a subsequent

immediate alert from the contact manager. This would occur, for example, if the behavior set its `completed_dist` to 100, but the contact manager was configured as it is in Listing 1 above, with the `alert_range` set to 80.

Without dynamic alert configurations in `pBasicContactMgr`, the configuration of the AvoidCollision behavior would need to be explicitly coordinated in two places, in the behavior configuration file and the mission configuration file. For example, one would need to make sure that line 8 in Listing 2 below reflected a completed distance *less than* the alert distance specified in line 9 in Listing 1 above.

Listing 7.2: Configuration snippet for AvoidCollision behavior.

```
1 //-----
2 Behavior = BHV_AvoidCollision
3 {
4     . . .
5     updates      = CONTACT_INFO
6     endflag      = CONTACT_RESOLVED = ${CONTACT}
7     templating   = spawn
8     completed_dist = 100
9     . . .
10 }
```

From the perspective of the contact manager, it simply supports both styles of alert configurations, and the issue of how or whether a client chooses to request alerts is of no concern to the contact manager, or those who configure it. The issue of how alert requests are generated for the AvoidCollision behavior is discussed separately in Section ??.

7.2.3 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 45.

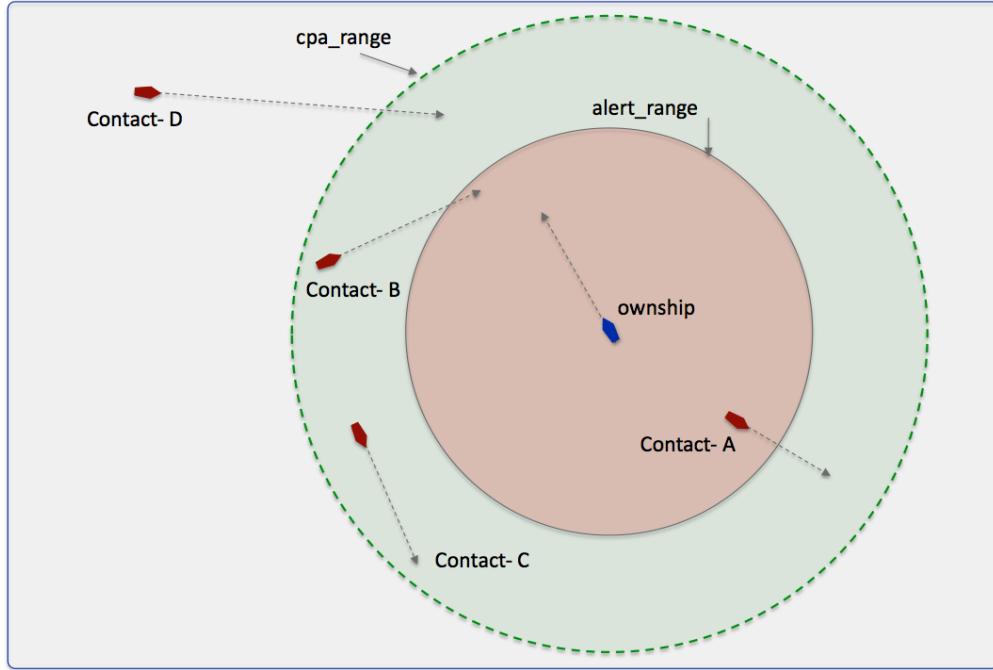


Figure 45: **Alert Triggers in pBasicContactMgr:** An alert may be triggered by `pBasicContactMgr` if the contact is within the `alert_range`, as with Contact-A. It may also be triggered if the contact is within the `cpa_range`, and the contact's CPA distance is within the `alert_range`, as with Contact-B. 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 45, 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.

7.2.4 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_RETIRE`:

```
CONTACTS_RETIRE = "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.

7.2.5 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.

7.3 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.)

7.4 Usage of the pBasicContactMgr with the IvP Helm

The IvP helm may be 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 receive 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 7.2.5. 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 ownership 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.

7.5 Terminal and AppCast Output

The status of the contact manager may be monitored from an open console window where `pBasicContactMgr` is launched. Example output is shown below in Listing 3.

Listing 7.3: 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 7.2.4. 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.

7.6 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 7.4: Configuration Parameters for `pBasicContactMgr`.

`alert:` A description of a single alert. Section 7.2.1.

<code>contact_local_coords:</code>	Determines if the local coordinates of incoming node reports are filled by translated latitude longitude coordinates. Legal values: <code>verbatim</code> , <code>lazy_lat_lon</code> , <code>force_lat_lon</code> . The default is <code>verbatim</code> , meaning no translation action is taken.
<code>default_alert_range:</code>	The range to a contact, in meters, within which an alert is posted. Legal values: any positive number. The default is 1000. Section 7.2.1 .
<code>default_cpa_range:</code>	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 7.2.1 .
<code>default_alert_range_color:</code>	The default color for rendering the alert range radius. Legal values: any color in Appendix B. The default is <code>gray70</code> . Section 7.2.1 .
<code>default_cpa_range_color:</code>	The default color for rendering the cpa range radius. Legal values: any color in Appendix B. The default is <code>gray30</code> . Section 7.2.1 .
<code>contact_max_age:</code>	Seconds between reports before a contact is dropped from the list. Legal values: any non-negative number. The default is 3600. Section 7.2.4 .
<code>display_radii:</code>	If true, the two alert ranges are posted as viewable circles. Legal values: true, false. The default is false.

7.6.1 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 5 below.

Listing 7.5: Example configuration of the pBasicContactMgr application.

```

1 =====
2 pBasicContactMgr Example MOOS Configuration
3 =====
4
5 ProcessConfig = pBasicContactMgr
6 {
7     AppTick    = 4
8     CommsTick = 4
9
10 // Alert configurations (one or more, keyed by id)
11 alert = id=avd, var=CONTACT_INFO
12 alert = id=avd, val="name=avd_${VNAME} # contact=${VNAME}"
13 alert = id=avd, range=80, alert_range_color=white
14 alert = id=avd, cpa_range=95, cpa_range_color=gray50
15
16 // Properties for all alerts

```

```

17 default_alert_range      = 1000 // the default in meters
18 default_cpa_range        = 1000 // the default in meters
19 default_alert_range_color = color // the default is gray65
20 default_cpa_range_color  = color // the default is gray35
21
22 // Policy for retaining potential stale contacts
23 contact_max_age         = 3600 // the default in secs.
24
25 // Configuring other output
26 display_radii            = false // or {true}
27 alert_verbose             = false // If true, ALERT_VERBOSE published.
28
29 // Policy for linear extrapolation of stale contacts
30 decay = 30,60 // the default in secs
31
32 contacts_recap_interval = 5 // the default in secs
33
34 contact_local_coords     = verbatim // the default
35 }

```

7.7 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
```

7.7.1 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 7.5.
- **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_RETIRE**: 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_RETIRE    = 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

```

7.7.2 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 appcast report, with reporting criteria, and expiration. Section 11.10.4.
- `BCM_DISPLAY_RADII`: If false, no postings will be made for rendering the alert and cpa range circles.
- `BCM_ALERT_REQUEST`: 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_HEADING`: Present ownship heading in degrees.
- `NAV_SPEED`: Present ownship speed in meters per second.
- `NAV_X`: Present position of ownship in local *x* coordinates.
- `NAV_Y`: Present position of ownship in local *y* coordinates.
- `NODE_REPORT`: A report about a known contact.

7.7.3 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 or -h
```

This will show the output shown in Listing 6 below.

Listing 7.6: 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.
```

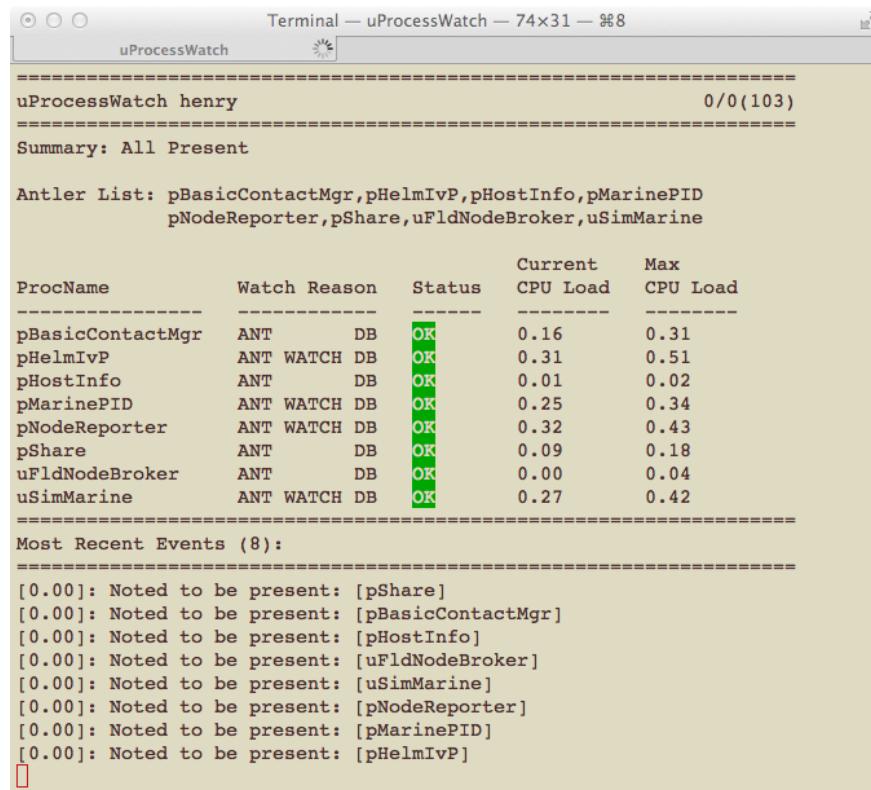
8 uProcessWatch: Monitoring MOOS Application Health

8.1 Overview

The uProcessWatch application monitors the health of a set of MOOS application. It does two things:

- It monitors the presence of a set of MOOS apps,
- It monitors the CPU load of a set of MOOS apps.

In the case of the former, `uProcessWatch` continually monitors the `DB_CLIENTS` list for applications it has responsibility for watching. In the case of the latter, MOOS apps are already monitoring and reporting their own CPU load and `uProcessWatch` is doing nothing more than gathering that information for display in the terminal or appcast message. An example terminal output is shown below:



The screenshot shows a terminal window titled "uProcessWatch" with the command "uProcessWatch henry" run. The output is as follows:

```
Terminal — uProcessWatch — 74x31 — %88
uProcessWatch henry                                         0/0(103)
=====
Summary: All Present

Antler List: pBasicContactMgr,pHelmIvP,pHostInfo,pMarinePID
              pNodeReporter,pShare,uFldNodeBroker,uSimMarine

      ProcName      Watch Reason      Status    Current      Max
      -----      -----      -----
pBasicContactMgr  ANT      DB      OK       0.16       0.31
pHelmIvP          ANT WATCH DB      OK       0.31       0.51
pHostInfo          ANT      DB      OK       0.01       0.02
pMarinePID         ANT WATCH DB      OK       0.25       0.34
pNodeReporter      ANT WATCH DB      OK       0.32       0.43
pShare             ANT      DB      OK       0.09       0.18
uFldNodeBroker     ANT      DB      OK       0.00       0.04
uSimMarine         ANT WATCH DB      OK       0.27       0.42
=====
Most Recent Events (8):
=====
[0.00]: Noted to be present: [pShare]
[0.00]: Noted to be present: [pBasicContactMgr]
[0.00]: Noted to be present: [pHostInfo]
[0.00]: Noted to be present: [uFldNodeBroker]
[0.00]: Noted to be present: [uSimMarine]
[0.00]: Noted to be present: [pNodeReporter]
[0.00]: Noted to be present: [pMarinePID]
[0.00]: Noted to be present: [pHelmIvP]
```

Figure 46: A typical terminal or appcast report for `uProcessWatch`.

The first line, "Summary: All Present", tells you that no processes are missing. This is also posted in the variable `PROC_WATCH_SUMMARY`. The list of applications launched with `pAntler` is shown in the next block. The main body shows, for each watched application, the reason why the process is on the watch list, the status, current CPU load as reported by the process itself, and the maximum CPU load noted so far.

The bottom section of the terminal output shows the events (similar to all appcasting output). In the case of `uProcessWatch`, events note either the arrival or disappearance of a watched process. Each event also results in the posting of the variable `PROC_WATCH_EVENT`. The user may configure `uProcessWatch` to *not* watch certain named applications or patterns of applications, such as `uXMS*`, to avoid unwarranted alerts.

8.2 Typical uProcessWatch Usage Scenarios

8.2.1 Using uProcessWatch with AppCasting and pMarineViewer

The first usage scenario is perhaps the most typical since it is viable both in simulation and in the field where the vehicles have a network connection to a shoreside MOOS community. The idea is shown in Figure 47 below. Multiple vehicles each run `uProcessWatch` locally. A network connection from each vehicle to a shoreside MOOS community is used to share appcast messages using `pShare`. The shoreside community is running `pMarineViewer` which supports a multi-vehicle appcast viewing mode. In this way, the shoreside user may monitor the health of all vehicles' applications with one tool. If a process on a single vehicle goes missing, the menu item on the shoreside viewer for that vehicle turns red.

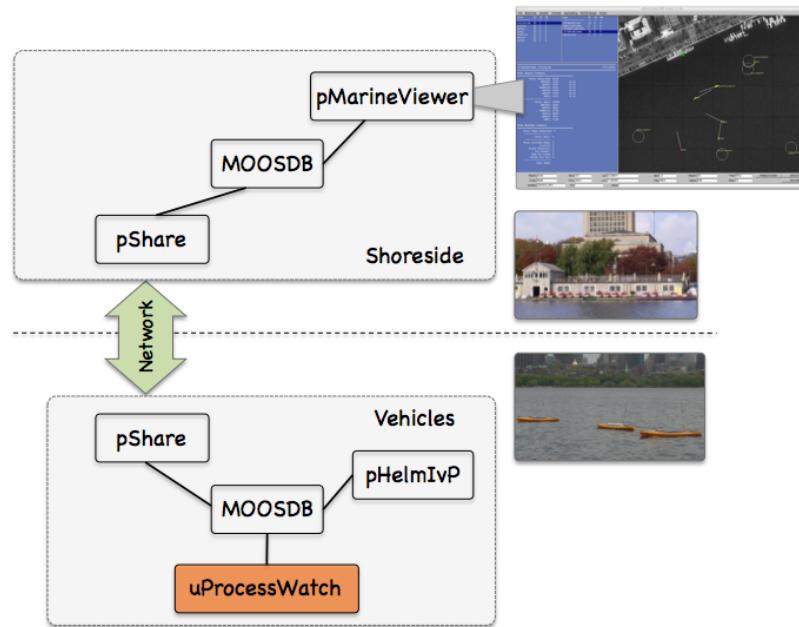


Figure 47: Using `uProcessWatch` with AppCasting: `uProcessWatch` is run locally on each fielded vehicle, generating appcasts posted to the local MOOSDB. Appcasts are shared to the shoreside and collected by the `pMarineViewer` tool for monitoring processes across all vehicle. This scenario is relevant when there is a network connection from the vehicles to the shoreside.

8.2.2 Directly Accessing the `PROC_WATCH_SUMMARY` Output

The main health indicator produced by `uProcessWatch` in the MOOS variable `PROC_WATCH_SUMMARY`. This can be checked on a remote machine by simply scoping on this variable. The `uMS` application

distributed with MOOS will do the trick for example. To focus solely on this variable, the `uXMS` tool may also be used as follows:

```
$ uXMS --server_host=10.25.0.72 --server_port=9000 PROC_WATCH_SUMMARY
```

Or one may ssh onto the vehicle and launch a scope locally on this variable.

8.3 Using and Configuring the uProcessWatch Utility

The primary configuration of `uProcessWatch` is defining the *watch list*, the set of other MOOS processes to monitor. By default all processes ever noted to be connected to the MOOSDB are on the watch list. The same with all processes name in the Antler configuration block of the mission file. This default is used because it is simple, and a good rule of thumb is that if any process disconnects from the MOOSDB, it's probably a sign of trouble.

8.3.1 The `DB_CLIENTS` Variable for Detecting Missing Processes

The MOOSDB, about once per second, posts the variable `DB_CLIENTS` containing a comma-separated list of clients (other MOOS apps) currently connected to the MOOSDB. When `uProcessWatch` is configured to watch for all processes, it simply augments the watch list for any process that ever appears on the incoming `DB_CLIENTS` mail. If the process is then missing at a later reading of this `DB_CLIENTS` mail, is is considered AWOL.

8.3.2 Defining the Watch List

The *watch list* is a list of processes, i.e., MOOS apps. Each item on the list will be reported missing if it does not appear in the list of clients shown by the current value of `DB_CLIENTS`. By default, all clients that ever appear on the `DB_CLIENTS` list will be added to the watch list. The same is true for all processes named in the Antler configuration block of the mission file. This default can be overridden by the following configuration option:

```
watch_all = false
```

The value of `watch_all` may also be set to "antler" to indicate that items on the Antler list are to be watched by default, but not those that appear in the `DB_CLIENTS` list. Likewise it may be set to "dbclients" to indicate that items in the `DB_CLIENTS` list are to be watched by default, but not those in the Antler list. Processes may be added to the watch list by explicitly naming them in configuration block as follows:

```
watch = process_name[*]
```

A process may be named explicitly, or the prefix of the process may be given, e.g., `watch=uXMS*`. This will match all processes fitting this pattern, e.g., `uXMS_845`, `uXMS_23`.

Processes may be explicitly *excluded* from the watch list with configurations of the following:

```
nowatch = process_name[*]
```

This is perhaps most appropriate when coupled with `watch_all=true`. If a process is for some reason both explicitly included and excluded, the inclusion takes precedent.

8.3.3 Reports Generated

There are three reports generated in the variables:

- `PROC_WATCH_EVENT`
- `PROC_WATCH_SUMMARY`,
- `PROC_WATCH_FULL_SUMMARY`

All reports are generated only when there is a change of status in one of the watched processes. The first type of report is generated for each event when a watched process is noted to have connected or disconnected to the MOOSDB. The following are examples:

```
PROC_WATCH_EVENT = "Process [pMarinePID] is noted to be present."
PROC_WATCH_EVENT = "Process [pMarinePID] has died!!!!"
PROC_WATCH_EVENT = "Process [pMarinePID] is resurrected!!!"
```

In the first line above, a process is reported to be present that was never previously on the watch list. In the third line a process that was previously noted to have left the watch list is reported to have returned or been restarted.

The `PROC_WATCH_SUMMARY` variable will list the set of processes missing from the watch list or report "All Present" if no items are missing. For example:

```
PROC_WATCH_SUMMARY = "All Present"
PROC_WATCH_SUMMARY = "AWOL: pMarinePID,uSimMarine"
```

The `PROC_WATCH_FULL_SUMMARY` variable will list a more complete and historical status for all processes on the watch list. For example:

```
PROC_WATCH_FULL_SUMMARY = "pHelmIvP(1/0),uSimMarine(1/1),pMarinePID(2/2)"
```

The numbers in parentheses indicate how many times the process has been noted to connect to the MOOSDB over the number of times it has been noted to have disconnected. A report of "(1/0)" is the healthiest of possible reports, meaning it has connected once and has never disconnected.

8.3.4 Watching and Reporting on a Single MOOS Process

If desired, `uProcessWatch` may be configured to generate a report dedicated a single MOOS process with the following example configuration:

```
watch = pBasicContactMgr : BCM_OK
```

In this case a MOOS variable, `BCM_OK`, will be set to either true or false depending on whether the process `pBasicContactMgr` presently appears on the list of connected clients in listed in `DB_CLIENTS`.

8.3.5 A Heartbeat for the Watch Dog

By default `uProcessWatch` will only post `PROC_WATCH_SUMMARY` when its value changes. A long stale `PROC_WATCH_SUMMARY = "All Present"` likely means that everything is fine. It could also mean the `uProcWatchSummary` process itself died without ever posting anything to suggest otherwise. The user has the configuration option to post `PROC_WATCH_SUMMARY` every N seconds regardless of whether or not the value has changed:

```
summary_wait = 30 // Summary posted at least every 30 seconds
```

This in effect creates a heartbeat for monitoring `uProcessWatch`. The default value for this parameter is -1 . Any negative value will be interpreted as a request for postings to be made only when the posting value changes. Regardless of the `summary_wait` setting, the other two reports, `PROC_WATCH_EVENT` and `PROC_WATCH_FULL_SUMMARY`, will only be made when the values change.

8.3.6 Excusing a Process

An application on the watch list may be excused and disconnect from the MOOSDB if it posts to the variable `EXITED_NORMALLY` with the name of itself. A check is made by `uProcessWatch` of the *source* of the posting to ensure that message was posted by the exiting process. It's up to the developer of an application to build in the feature declaring a normal exit. An example is the `uTimerScript` application described in Section 6, which has the ability to execute a script of postings to the MOOSDB followed by an exit. In this case, the *status* of application becomes EXCUSED, as shown in Figure 48.

ProcName	Watch Reason	Status	Current CPU Load	Max CPU Load
pHelmIvP	ANT	DB	0.85	1.67
pLogger	ANT	DB	1.38	1.79
pMarinePID	ANT	DB	0.85	1.18
pMarineViewer	ANT	DB	6.47	7.45
pNodeReporter	ANT	DB	0.19	0.38
uSimMarine	ANT	DB	0.45	0.78
uTimerScript	ANT	DB	1.53	1.53

Most Recent Events (8):

```
[10.06]: PROC_WATCH_EVENT: Process [uTimerScript] is gone but excused.  
[0.00]: Noted to be present: [pLogger]  
[0.00]: Noted to be present: [uSimMarine]  
[0.00]: Noted to be present: [pMarinePID]  
[0.00]: Noted to be present: [pHelmIvP]  
[0.00]: Noted to be present: [pMarineViewer]  
[0.00]: Noted to be present: [pNodeReporter]  
[0.00]: Noted to be present: [uTimerScript]
```

Figure 48: The process `uTimerScript` has gone missing due to its own intentional exit. Before exiting it posted `EXITED_NORMALLY=uTimerScript`, and therefore `uTimerScript` is regarded as excused.

Excusing a missing process is different than choosing to not include it on the watch list. Some applications are, by their nature, designed to disappear at some point. Scoping tools like [uXMS](#), [uPokeDB](#) or [uHelmScope](#) are examples applications that regularly appear and disappear. They are best excluded entirely from the watch list with the `nowatch` configuration parameter.

8.3.7 Allowing Retractions if a Process Reappears

With the addition of appcasting to [uProcessWatch](#), a missing process also triggers an appcast run warning, as shown on the top in Figure 49 below.

```

Terminal — uProcessWatch — 77x36 — %6
uMAC uProcessWatch
=====
uProcessWatch henry 0/1(269)
=====
Runtime Warnings: 1
[1]: Process [pNodeReporter] is missing.

Summary: AWOL: pNodeReporter

Antler List: pBasicContactMgr,pHelmIpv,pHostInfo,pMarinePID
             pNodeReporter,pShare,uFldMessageHandler,uFldNodeBroker
             uSimMarine

ProcName      Watch Reason   Status    Current CPU Load  Max CPU Load
-----        -----          -----      -----      -----
pBasicContactMgr ANT DB      OK        0.13       0.16
pHelmIpv      ANT WATCH DB  OK        0.12       0.28
pHostInfo     ANT DB      OK        0.01       0.02
pMarinePID    ANT WATCH DB  OK        0.21       0.25
pNodeReporter ANT WATCH DB  MISSING   0.27       0.28
pShare         ANT DB      OK        0.06       0.12
uFldMessageHandler ANT DB   OK        0.01       0.04
uFldNodeBroker ANT DB      OK        0.01       0.04
uSimMarine    ANT WATCH DB  OK        0.30       0.34
=====

Most Recent Events (8):
=====
[149.17]: PROC_WATCH_EVENT: Process [pNodeReporter] is missing.
[0.00]: Noted to be present: [pShare]
[0.00]: Noted to be present: [pBasicContactMgr]
[0.00]: Noted to be present: [pHostInfo]
[0.00]: Noted to be present: [uFldNodeBroker]
[0.00]: Noted to be present: [uFldMessageHandler]
[0.00]: Noted to be present: [uSimMarine]
[0.00]: Noted to be present: [pNodeReporter]

```

Figure 49: The process pNodeReporter has gone missing. This is noted as a runtime warning at the top, and the MISSING status in the body of the report.

However, there are cases where a process goes missing but then returns, in which case the warning is a distraction. These reasons include.

- The application may actually exit and then restart a short time later.
- The application may be running at a very slow apptick in which case it may be dropped from the `DB_CLIENTS` list momentarily.
- The application may be missing only because it hasn't launched yet.

To address this, [uProcessWatch](#) will allow retractions on appcast run warnings. If the application disappears and reappears, the appcast run warning will disappear. The successive posted values of `PROC_WATCH_SUMMARY` will show that the process was at some point declared AWOL, but once the process returns, the appcast output will no longer raise attention to the issue.

Of course there are situations where a process that disappears and then reappears is an indication of a problem, not to be swept under the rug. In this case the default behavior of `uProcessWatch` may be changed by setting `allow_retractions` to false. Presently this setting cannot be set on a per-process manner.

8.4 Configuration Parameters of `uProcessWatch`

The following parameters are defined for `uProcessWatch`. A more detailed description is provided in other parts of this section. Parameters having default values indicate so in parentheses below.

Listing 8.1: Configuration Parameters for `uProcessWatch`.

- `allow_retractions`: If true, run warnings are retracted if a process reappears after disappearing.
Legal values: true, false. The default is true.
- `nowatch`: A process or list of MOOS processes to *not* watch.
- `post_mapping`: A mapping from one posting variable name to another.
- `summary_wait`: A maximum amount of time between `PROC_WATCH_SUMMARY` postings. Negative value indicates posting occurs only when the value changes regardless of elapsed time. Legal values: any numerical value. The default is `-1`.
- `watch`: One or more comma-separated MOOS process to watch and report on.
- `watch_all`: If true, watch all processes that become known either via the `DB_CLIENTS` list or the Antler list. Legal values: true, false. The default is true.

8.4.1 An Example MOOS Configuration Block

Listing 2 shows an example MOOS configuration block produced from the following command line invocation:

```
$ uProcessWatch --example or -e
```

Listing 8.2: Example configuration of the `uProcessWatch` application.

```

1 ProcessConfig = uProcessWatch
2 {
3     AppTick    = 4
4     CommsTick = 4
5
6     watch_all = true // The default is true.
7
8     watch    = pMarinePID:PID_OK
9     watch    = uSimMarine:USM_OK
10
11    nowatch = uXMS*
12
```

```

13     allow_retractions = true    // Always allow run-warnings to be
14                     // retracted if proc re-appears
15
16     // A negative value means summary only when status changes.
17     summary_wait = 10 // Seconds. Default is -1.
18
19     post_mapping = PROC_WATCH_FULL_SUMMARY, UPW_FULL_SUMMARY
20 }

```

8.5 Publications and Subscriptions for uProcessWatch

The interface for `uProcessWatch`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uProcessWatch --interface or -i
```

8.5.1 Variables Published by uProcessWatch

The primary output of `uProcessWatch` to the MOOSDB is a summary indicating whether or not certain other processes (MOOS apps) are presently connected.

- `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 11.10.4.
- `PROC_WATCH_EVENT`: A report indicating a particular process has been noted to be gone missing or noted to have (re)joined the list of active processes.
- `PROC_WATCH_FULL_SUMMARY`: A single string report for each process indicating how many times it has connected and disconnected from the MOOSDB.
- `PROC_WATCH_SUMMARY`: A report listing all missing processes, or "All Present" if no processes are missing.

The user may also configure `uProcessWatch` to make a posting dedicated to a particular watched process. For example, with the configuration `watch=pNodeReporter:PNR_OK`, the status of this process is conveyed in the MOOS variable `PNR_OK`, set to either true or false depending on whether or not it is present.

The variable name for any posted variable may be changed to a different name with the `post_mapping` configuration parameter. For example, `post_mapping=PROC_WATCH_EVENT, UPW_EVENT` will result in events being posted under the `UPW_EVENT` variable rather than `PROC_WATCH_EVENT` variable.

8.5.2 MOOS Variables Subscribed for by uProcessWatch

The following variable(s) will be subscribed for by `uProcessWatch`:

- `APPCAST_REQ`: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 11.10.4.
- `DB_CLIENTS`: A comma-separated list of clients currently connected to the MOOSDB, posted by the MOOSDB. Used for detecting missing processes. Section 8.3.1.

- **EXITED_NORMALLY**: An indication made by a process, potentially on the watch list, that it has exited normally and should be excused, and not regarded as missing. Section [8.3.6](#).
- **<PROCNAME>_STATUS**: The status string generated for all MOOSApps, containing the current CPU load among other things.

9 uSimMarine: Basic Vehicle Simulation

The `uSimMarine` application is a simple 3D vehicle simulator that updates vehicle state, position and trajectory, based on the present actuator values and prior vehicle state. The typical usage scenario has a single instance of `uSimMarine` associated with each simulated vehicle, as shown in Figure 50.

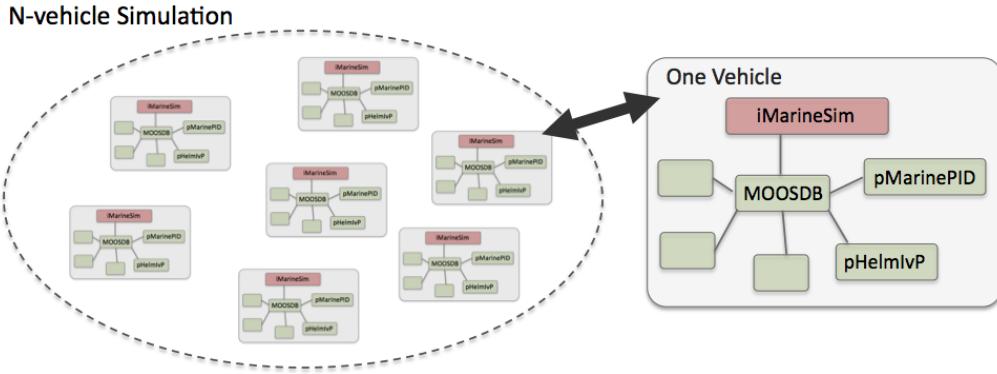


Figure 50: **Typical `uSimMarine` Usage:** In an N-vehicle simulation, an instance of `uSimMarine` is used for each vehicle. Each simulated vehicle typically has its own dedicated MOOS community. The IvP Helm (`pHelmIvP`) publishes high-level control decisions. The PID controller (`pMarinePID`) converts the high-level control decisions to low-level actuator decisions. Finally the simulator (`uSimMarine`) reads the low-level actuator postings to produce a new vehicle position.

This style of simulation can be contrasted with simulators that simulate a comprehensive set of aspects of the simulation, including multiple vehicles, and aspects of the environment and communications. The `uSimMarine` simulator simply focuses on a single vehicle. It subscribes for the vehicle navigation state variables `NAV_X`, `NAV_Y`, `NAV_SPEED`, `NAV_HEADING`, `NAV_DEPTH`, as well as the actuator values `DESIRED_RUDDER`, `DESIRED_THRUST`, `DESIRED_ELEVATOR`. The `uSimMarine` accommodates a notion of external drifts applied to the vehicle to crudely simulate current or wind. These drifts may be set statically or may be changing dynamically by other MOOS processes. The simulator also may be configured with a simple geo-referenced data structure representing a field of water currents.

Under typical UUV payload autonomy operation, the `uSimMarine` and `pMarinePID` MOOS modules would not be present. The vehicle's native controller would handle the role of `pMarinePID`, and the vehicle's native navigation system (and the vehicle itself) would handle the role of `uSimMarine`.

9.1 Configuration Parameters for `uSimMarine`

The following parameters are defined for `uSimMarine`. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so in parentheses below.

Listing 9.1: Configuration Parameters for `uSimMarine`.

- `buoyancy_rate`: Rate, in meters per second, at which vehicle floats to surface at zero speed. The default is zero. Section 9.4.4.
- `current_field`: A file containing the specification of a current field.

```

current_field_active: If true, simulator uses the current field if specified.
default_water_depth: Default value for local water depth for calculating altitude (0).
drift_vector: A pair of external drift values, direction and magnitude.
rotate_speed: An external rotational speed in degrees per second (0).
    drift_x: An external drift value applied in the x direction (0).
    drift_y: An external drift value applied in the y direction (0).
max_acceleration: Maximum rate of vehicle acceleration in  $m/s^2$  (0.5).
max_deceleration: Maximum rate of vehicle deceleration in  $m/s^2$  (0.5).
max_depth_rate: Maximum rate of vehicle depth change, meters per second. The default is 0.5. Section 9.4.4.
max_depth_rate_speed: Vehicle speed at which max depth rate is achievable (2.5). Section 9.4.4.
    prefix: Prefix of MOOS variables published. The default is USM..
    sim_pause: If true, the simulation is paused. The default is false.
start_depth: Initial vehicle depth in meters. The default is zero. Section 9.3.
start_heading: Initial vehicle heading in degrees. The default is zero. Section 9.3.
    start_pos: A full starting position and trajectory specification. Section 9.3.
start_speed: Initial vehicle speed in meters per second. The default is zero. Section 9.3.
    start_x: Initial vehicle x position in local coordinates. The default is zero. Section 9.3.
    start_y: Initial vehicle y position in local coordinates. The default is zero. Section 9.3.
thrust_factor: A scalar correlation between thrust and speed. The default is 20.
thrust_map: A mapping between thrust and speed values. Section 9.7.
thrust_reflect: If true, negative thrust is simply opposite positive thrust. The default is false. Section 9.7.
turn_loss: A range [0, 1] affecting speed lost during a turn. The default is 0.85.
turn_rate: A range [0, 100] affecting vehicle turn radius, e.g., 0 is an infinite turn radius. The default is 70.

```

An Example MOOS Configuration Block

An example MOOS configuration block is provided in Listing 2 below. This can also be obtained from a terminal window with:

```
$ uSimMarine --example or -e
```

Listing 9.2: Example configuration of the uSimMarine application.

```

1 =====
2 uSimMarine Example MOOS Configuration

```

```

3 =====
4
5 ProcessConfig = uSimMarine
6 {
7     AppTick    = 4
8     CommsTick = 4
9
10    start_x      = 0
11    start_y      = 0
12    start_heading = 0
13    start_speed   = 0
14    start_depth   = 0
15    start_pos     = x=0, y=0, speed=0, heading=0, depth=0
16
17    drift_x       = 0
18    drift_y       = 0
19    rotate_speed  = 0
20    drift_vector  = 0,0      // heading, magnitude
21
22    buoyancy_rate    = 0.025 // meters/sec
23    max_acceleration = 0      // meters/sec^2
24    max_deceleration = 0.5    // meters/sec^2
25    max_depth_rate   = 0.5    // meters/sec
26    max_depth_rate_speed = 2.0 // meters/sec
27
28    sim_pause        = false // or {true}
29    dual_state        = false // or {true}
30    thrust_reflect    = false // or {true}
31    thrust_factor     = 20    // range [0,inf)
32    turn_rate         = 70    // range [0,100]
33    thrust_map        = 0:0, 20:1, 40:2, 60:3, 80:5, 100:5
34 }

```

9.2 Publications and Subscriptions for uSimMarine

The interface for `uSimMarine`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ usimMarine --interface or -i
```

9.2.1 Variables Published by uSimMarine

The primary output of `uSimMarine` to the MOOSDB is the full specification of the updated vehicle position and trajectory, along with a few other pieces of information:

- **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.
- **BUOYANCY_REPORT**:
- **TRIM_REPORT**:
- **USM_ALTITUDE**: The updated vehicle altitude in meters if water depth known.
- **USM_DEPTH**: The updated vehicle depth in meters. Section 9.4.4.

- **USM_DRIFT_SUMMARY**: A summary of the current total external drift.
- **USM_HEADING**: The updated vehicle heading in degrees.
- **USM_HEADING_OVER_GROUND**: The updated vehicle heading over ground.
- **USM_LAT**: The updated vehicle latitude position.
- **USM_LONG**: The updated vehicle longitude position.
- **USM_RESET_COUNT**: The number of time the simulator has been reset.
- **USM_SPEED**: The updated vehicle speed in meters per second.
- **USM_SPEED_OVER_GROUND**: The updated speed over ground.
- **USM_X**: The updated vehicle *x* position in local coordinates.
- **USM_Y**: The updated vehicle *y* position in local coordinates.
- **USM_YAW**: The updated vehicle yaw in radians.

An example **USM_DRIFT_SUMMARY** string: "ang=90, mag=1.5, xmag=90, ymag=0".

9.2.2 Variables Subscribed for by uSimMarine

The **uSimMarine** application will subscribe for the following MOOS variables:

- **APPCAST_REQ**: A request to generate and post a new appcast report, with reporting criteria, and expiration. Section 11.10.4.
- **DESIRED_THRUST**: The thruster actuator setting, [-100, 100].
- **DESIRED_RUDDER**: The rudder actuator setting, [-100, 100].
- **DESIRED_ELEVATOR**: The depth elevator setting, [-100, 100].
- **USM_SIM_PAUSED**: Simulation pause request, either **true** or **false**.
- **USM_CURRENT_FIELD**: If **true**, a configured current field is active.
- **USM_BUOYANCY_RATE**: Dynamically set the zero-speed float rate.
- **ROTATE_SPEED**: Dynamically set the external rotational speed.
- **DRIFT_X**: Dynamically set the external drift in the *x* direction.
- **DRIFT_Y**: Dynamically set the external drift in the *y* direction.
- **DRIFT_VECTOR**: Dynamically set the external drift direction and magnitude.
- **DRIFT_VECTOR_ADD**: Dynamically modify the external drift vector.
- **DRIFT_VECTOR_MULT**: Dynamically modify the external drift vector magnitude.
- **USM_RESET**: Reset the simulator with a new position, heading, speed and depth.
- **WATER_DEPTH**: Water depth at the present vehicle position.

Each iteration, after noting the changes in the navigation and actuator values, it posts a new set of navigation state variables in the form of **USM_X**, **USM_Y**, **USM_SPEED**, **USM_HEADING**, **USM_DEPTH**.

9.2.3 Command Line Usage of uSimMarine

The **uSimMarine** 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. The basic command line usage for the **uSimMarine** application is the following:

Listing 9.3: Command line usage for the uSimMarine application.

```

1 Usage: uSimMarine file.moos [OPTIONS]
2
3 Options:
4   --alias=<ProcessName>
5       Launch uSimMarine with the given process name
6       rather than uSimMarine.
7   --example, -e
8       Display example MOOS configuration block.
9   --help, -h
10      Display this help message.
11   --version,-v
12      Display the release version of uSimMarine.

```

9.3 Setting the Initial Vehicle Position, Pose and Trajectory

The simulator is typically configured with a vehicle starting position, pose and trajectory given by the following five configuration parameters:

- `start_x`
- `start_y`
- `start_heading`
- `start_speed`
- `start_depth`

The position is specified in local coordinates in relation to a local datum, or (0,0) position. This datum is specified in the `.moos` file at the global level. The heading is specified in degrees and corresponds to the direction the vehicle is pointing. The initial speed and depth by default are zero, and are often left unspecified in configuration. Alternatively, the same five parameters may be set with the `start_pos` parameter as follows:

```
start_pos = x=100, y=150, speed=0, heading=45, depth=0
```

The simulator can also be reset at any point during its operation, by posting to the MOOS variable `USM_RESET`. A posting of the following form will reset the same five parameters as above:

```
USM_RESET = x=200, y=250, speed=0.4, heading=135, depth=10
```

This has been useful in cases where the objective is to observe the behavior of a vehicle from several different starting positions, and an external MOOS script, e.g., `uTimerScript`, is used to reset the simulator from each of the desired starting states.

9.4 Propagating the Vehicle Speed, Heading, Position and Depth

The vehicle position is updated on each iteration of the `uSimMarine` application, based on (a) the previous vehicle state, (b) the elapsed time since the last update, ΔT , (c) the current actuator values, `DESIRED_RUDDER`, `DESIRED_THRUST`, and `DESIRED_ELEVATOR`, and (d) several parameter settings describing the vehicle model.

For simplicity, this simulator updates the vehicle speed, heading, position and depth in sequence, in this order. For example, the position is updated after the heading is updated, and the new

position update is made as if the new heading were the vehicle heading for the entire ΔT . The error introduced by this simplification is mitigated by running `uSimMarine` with a fairly high MOOS AppTick value keeping the value of ΔT sufficiently small.

9.4.1 Propagating the Vehicle Speed

The vehicle speed is propagated primarily based on the current value of thrust, which is presumably refreshed upon each iteration by reading the incoming mail on the MOOS variable `DESIRED_THRUST`. To simulate a small speed penalty when the vehicle is conducting a turn through the water, the new thrust value may also be affected by the current rudder value, referenced by the incoming MOOS variable `DESIRED_RUDDER`. The newly calculated speed is also dependent on the previously noted speed noted by the incoming MOOS variable `NAV_SPEED`, and the settings to the two configuration parameters `MAX_ACCELERATION` and `MAX_DECELERATION`.

The algorithm for updating the vehicle speed proceeds as:

1. Calculate $v_{i(\text{RAW})}$, the new raw speed based on the thrust.
2. Calculate $v_{i(\text{TURN})}$, an adjusted and potentially lower speed, based on the raw speed, $v_{i(\text{RAW})}$, and the current rudder angle, `DESIRED_RUDDER`.
3. Calculate $v_{i(\text{FINAL})}$, an adjusted and potentially lower speed based on $v_{i(\text{TURN})}$, compared to the prior speed. If the magnitude of change violates either the max acceleration or max deceleration settings, then the new speed is clipped appropriately.
4. Set the new speed to be $v_{i(\text{FINAL})}$, and use this new speed in the later updates on heading, position and depth.

Step 1: In the first step, the new speed is calculated by the current value of thrust. In this case the *thrust map* is consulted, which is a mapping from possible thrust values to speed values. The thrust map is configured with the `THRUST_MAP` configuration parameter, and is described in detail in Section 9.7.

$$v_{i(\text{RAW})} = \text{THRUST_MAP}(\text{DESIRED_THRUST})$$

Step 2: In the second step, the calculated speed is potentially reduced depending on the degree to which the vehicle is turning, as indicated by the current value of the MOOS variable `DESIRED_RUDDER`. If it is not turning, it is not diminished at all. The adjusted speed value is set according to:

$$v_{i(\text{TURN})} = v_{i(\text{RAW})} * \left(1 - \left(\frac{|\text{RUDDER}|}{100} * \text{TURN_LOSS}\right)\right)$$

The configuration parameter `turn_loss` is a value in the range of $[0, 1]$. When set to zero, there is no speed lost in any turn. When set to 1, there is a 100% speed loss when there is a maximum rudder. The default value is 0.85.

Step 3: In the last step, the candidate new speed, $v_{i(\text{TURN})}$, is compared with the incoming vehicle speed, v_{i-1} . The elapsed time since the previous simulator iteration, ΔT , is used to calculate the acceleration or deceleration implied by the new speed. If the change in speed violates either the `min_acceleration`, or `max_acceleration` parameters, the speed is adjusted as follows:

$$v_{i(\text{FINAL})} = \begin{cases} v_{i-1} + (\text{MAX_ACCELERATION} * \Delta T) & \frac{(v_{i(\text{TURN})} - v_{i-1})}{\Delta T} > \text{MAX_ACCELERATION}, \\ v_{i-1} - (\text{MAX_DECELERATION} * \Delta T) & \frac{(v_{i-1} - v_{i(\text{TURN})})}{\Delta T} > \text{MAX_DECELERATION}, \\ v_{i(\text{TURN})} & \text{otherwise.} \end{cases}$$

Step 4: The final speed from the previous step is posted by the simulator as `USM_SPEED`, and is used the calculations of position and depth, described next.

9.4.2 Propagating the Vehicle Heading

The vehicle heading is propagated primarily based on the current `RUDDER` value which is refreshed upon each iteration by reading the incoming mail on the MOOS variable `DESIRED_RUDDER`, and the elapsed time since the simulator previously updated the vehicle state, ΔT . The change in heading may also be influenced by the `THRUST` value from the MOOS variable `DESIRED_THRUST`, and may also factor an external rotational speed.

The algorithm for updating the new vehicle heading proceeds as:

1. Calculate $\Delta\theta_{i(\text{RAW})}$, the new raw change in heading influenced only by the current rudder value.
2. Calculate $\Delta\theta_{i(\text{THRUST})}$, an adjusted change in heading, based on the raw change in heading, $\Delta\theta_{i(\text{RAW})}$, and the current `THRUST` value.
3. Calculate $\Delta\theta_{i(\text{EXTERNAL})}$, an adjusted change in heading considering external rotational speed.
4. Calculate θ_i , the final new heading based on the calculated change in heading and the previous heading, and converted to the range of [0, 359].

Step 1: In the first step, the new heading is calculated by the current `RUDDER` value:

$$\Delta\theta_{i(\text{RAW})} = \text{RUDDER} * \frac{\text{TURN_RATE}}{100} * \Delta T$$

The `TURN_RATE` is an `uSimMarine` configuration parameter with the allowable range of [0, 100]. The default value of this parameter is 70, chosen in part to be consistent with the performance of the simulator prior to this parameter being exposed to configuration. A value of 0 would result in the vehicle never turning, regardless of the rudder value.

Step 2: In the second step the influence of the current vehicle thrust (from the MOOS variable `DESIRED_THRUST`) may be applied to the change in heading. The magnitude of the change of heading is adjusted to be greater when the thrust is greater than 50% and less when the thrust is less than 50%.

$$\Delta\theta_{i(\text{THRUST})} = \theta_{i(\text{RAW})} * (1 + \frac{|\text{THRUST}| - 50}{50})$$

The direction in heading change is then potentially altered based on the sign of the `THRUST`:

$$\Delta\theta_{i(\text{THRUST})} = \begin{cases} -\Delta\theta_{i(\text{THRUST})} & \text{THRUST} < 0, \\ \Delta\theta_{i(\text{THRUST})} & \text{otherwise.} \end{cases}$$

Step 3: In the third step, the change in heading may be further influenced by an external rotational speed. This speed, if present, would be read at the outset of the simulator iteration from either the configuration parameter `rotate_speed`, or dynamically from the MOOS variable `ROTATE_SPEED`. The updated value is calculated as follows:

$$\Delta\theta_{i(\text{EXTERNAL})} = \theta_{i(\text{THRUST})} + (\text{ROTATE_SPEED} * \Delta T)$$

Step 4: In final step, the final new heading is set based on the previous heading and the change in heading calculated in the previous three steps. If needed, the value of the new heading is converted to its equivalent heading in the range [0, 359].

$$\theta_i = \text{heading360}(\theta_{i-1} + \Delta\theta_{i(\text{EXTERNAL})})$$

The simulator then posts this value to the MOOSDB as `USM_HEADING`.

9.4.3 Propagating the Vehicle Position

The vehicle position is propagated primarily based on the newly calculated vehicle heading and speed, the previous vehicle position, and the elapsed time since updating the previous vehicle position, ΔT .

The algorithm for updating the new vehicle position proceeds as:

1. Calculate the vehicle heading and speed used for updating the new vehicle position, with the heading converted into radians.
2. Calculate the new positions, x_i and y_i , based on the heading, speed and elapsed time.
3. Calculate a possibly revised new position, factoring in any external drift.

Step 1: In the first step, the heading value, $\bar{\theta}$, and speed value, \bar{v} used for calculating the new vehicle position is set averaging the newly calculated values with their prior values:

$$\bar{v} = \frac{(v_i + v_{i-1})}{2} \quad (1)$$

$$\bar{\theta} = \text{atan2}(s, c)$$

where s and c are given by:

$$\begin{aligned} s &= \sin(\theta_{i-1}\pi/180) + \sin(\theta_i\pi/180) \\ c &= \cos(\theta_{i-1}\pi/180) + \cos(\theta_i\pi/180) \end{aligned}$$

The above calculation of the heading average handles the issue of angle wrap, i.e., the average of 359 and 1 is zero, not 180.

Step 2: The vehicle x and y position is updated by the following two equations:

$$x_i = x_{i-1} + \sin(\bar{\theta}) * \bar{v} * \Delta T$$

$$y_i = y_{i-1} + \cos(\bar{\theta}) * \bar{v} * \Delta T$$

The above is calculated keeping in mind the difference in convention used in marine navigation where zero degrees is due North and 90 degrees is due East. That is, the mapping is as follows from marine to traditional trigonometric convention: $0^\circ \rightarrow 90^\circ$, $90^\circ \rightarrow 0^\circ$, $180^\circ \rightarrow 270^\circ$, $270^\circ \rightarrow 180^\circ$.

Step 3: The final step adjusts the x , and y position from above, taking into consideration any external drift that may be present. This drift includes both the drift that may be directed from the incoming MOOS variables as described in Section 9.6. The drift components below are also a misnomer since they are provided in units of meters per second.

$$x_i = x_i + \text{EXTERNAL_DRIFT_X} * \Delta T \quad (2)$$

$$y_i = y_i + \text{EXTERNAL_DRIFT_Y} * \Delta T \quad (3)$$

9.4.4 Propagating the Vehicle Depth

Depth change in `uSimMarine` is simulated based on a few input parameters. The primary parameter that changes from one iteration to the next is the `ELEVATOR` actuator value, from the MOOS variable `DESIRED_ELEVATOR`. On any given iteration the new vehicle depth, z_i , is determined by:

$$z_i = z_{i-1} + (\dot{z}_i * \Delta t)$$

The new vehicle depth is altered by the *depth change rate*, \dot{z}_i , applied to the elapsed time, Δt , which is roughly equivalent to the `apptick` interval set in the `uSimMarine` configuration block. The depth change rate on the current iteration is determined by the vehicle speed as set in (1) and the `ELEVATOR` actuator value, and by the following three vehicle-specific simulator configuration parameters that allow for some variation in simulating the physical properties of the vehicle. The `buoyancy_rate`, for simplicity, is given in meters per second where positive values represent a positively buoyant vehicle. The `max_depth_rate`, and `max_depth_rate_speed` parameters determine the function(s) shown in Figure 51. The vehicle will have a higher depth change rate at higher speeds, up to some maximum speed where the speed no longer affects the depth change rate. The actual depth change rate then depends on the elevator and vehicle speed.

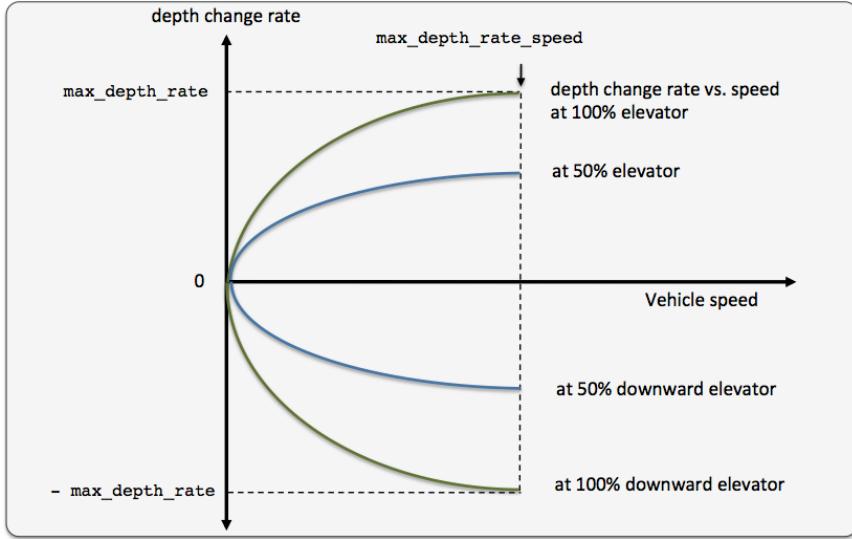


Figure 51: The relationship between the rate of depth change rate, given a current vehicle speed. Different elevator settings determine unique curves as shown.

The value of the depth change rate, \dot{z}_i , is determined as follows:

$$\dot{z}_i = \left(\frac{\bar{v}}{\text{MAX_DEPTH_RATE_SPEED}} \right)^2 * \frac{\text{ELEVATOR}}{100} * \text{MAX_DEPTH_RATE} + \text{BUOYANCY_RATE} \quad (4)$$

Both fraction components in 4 are clipped to $[-1, 1]$. When the vehicle is in reverse thrust and has a negative speed, this equation still holds. However, a vehicle would likely not have a depth change rate curve symmetric between positive and negative vehicle speeds. By default the value of `buoyancy_rate` is set to 0.025, slightly positively buoyant, `max_depth_rate` is set to 0.5, and `max_depth_rate_speed` is set to 2.0. The prevailing buoyancy rate may be dynamically adjusted by a separate MOOS application publishing to the variable `BUOYANCY_RATE`.

9.5 Propagating the Vehicle Altitude

The vehicle altitude is base solely on the current vehicle depth and the depth of the water at the current vehicle position. If nothing is known about the water depth, then `USM_ALTITUDE` is not published. The simulator may be configured with a default water depth:

```
default_water_depth = 100
```

This will allow the simulator to produce some altitude information if needed for testing consumers of `USM_ALTITUDE` information. Furthermore, the simulator subscribes for water depth information in the variable `USM_WATER_DEPTH` which could conceivably be produced by another MOOS application with access to bathymetry data and the vehicle's navigation position.

9.6 Simulation of External Drift

When the simulator updates the vehicle position as in equations (2) and (3), it factors a possible external drift in the x and y directions, in the term `EXTERNAL_DRIFT_X`, and `EXTERNAL_DRIFT_Y` respectively. The external drift may have two distinct components; a drift applied generally, and a drift applied due to a current field configured with an external file correlating drift vectors to local x and y positions. These drifts may be set in one of three ways discussed next.

9.6.1 External X-Y Drift from Initial Simulator Configuration

An external drift may be configured upon startup by either specifying explicitly the drift in the x and y direction, or by specifying a drift magnitude and direction. Figure 52 shows two external drifts each with the appropriate configuration using either the `drift_x` and `drift_y` parameters or the single `drift_vector` parameter:

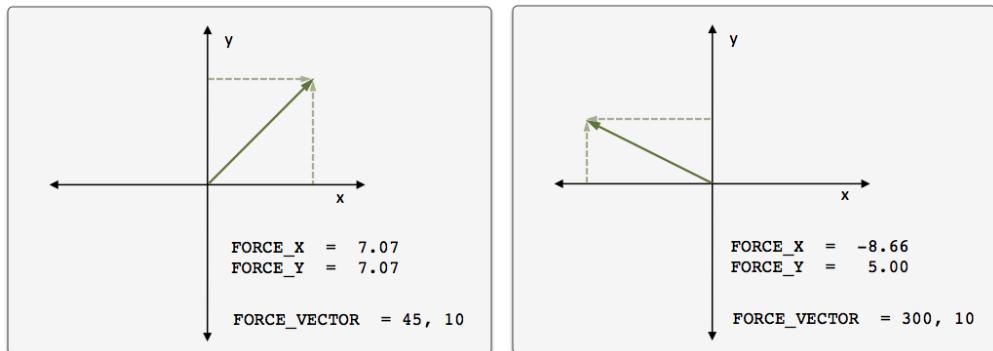


Figure 52: *External Drift Vectors*: Two drift vectors each configured with either the `drift_x` and `drift_y` configuration parameters or their equivalent single `drift_vector` parameter.

If, for some reason, the user mistakenly configures the simulator with both configuration styles, the configuration appearing last in the configuration block will be the prevailing configuration. If `uSimMarine` is configured with these parameters, these external drifts will be applied on the very first iteration and all later iterations unless changed dynamically, as discussed next.

9.6.2 External X-Y Drift Received from Other MOOS Applications

External drifts may be adjusted dynamically by other MOOS applications based on any criteria wished by the user and developer. The `uSimMarine` application registers for the following MOOS variables in this regard: `DRIFT_X`, `DRIFT_Y`, `DRIFT_VECTOR`, `DRIFT_VECTOR_ADD`, `DRIFT_VECTOR_MULT`. The first three variables simply override the previously prevailing drift, set by either the initial configuration or the last received mail concerning the drift.

By posting to the `USM_DRIFT_VECTOR_MULT` variable the *magnitude* of the prevailing vector may be modified with a single multiplier such as:

```
DRIFT_VECTOR_MULT = 2  
DRIFT_VECTOR_MULT = -1
```

The first MOOS posting above would double the size of the prevailing drift vector, and the second example would reverse the direction of the vector. The `DRIFT_VECTOR_ADD` variable describes a drift vector to be *added* to the prevailing drift vector. For example, consider the prevailing drift vector shown on the left in Figure 52, with the following MOOS mail received by the simulator:

```
DRIFT_VECTOR_ADD = "262.47, 15.796"
```

The resulting drift vector would be the vector shown on the right in Figure 52. This interface opens the door for the scripting changes to the drift vector like the one below, that crudely simulate a gust of wind in a given direction that builds up to a certain magnitude and dies back down to a net zero drift.

```
DRIFT_VECTOR_ADD = 137, 0.25
DRIFT_VECTOR_ADD = 137, -0.25
```

The above style script was described in the Section 6.9.2, where the `uTimerScript` utility was used to simulate wind gusts in random directions with random magnitude. The `DRIFT_*` interface may also be used by any third party MOOS application simulating things such as ocean or wind currents. The `uSimMarine` application does have native support for simple simulation with current fields as described next.

9.7 The ThrustMap Data Structure

A *thrust map* is a data structure that may be used to simulate a non-linear relationship between thrust and speed. This is configured in the `uSimMarine` configuration block with the `thrust_map` parameter containing a comma-separated list of colon-separated pairs. Each element in the comma-separated list is a single mapping component. In each component, the value to the left of the colon is a thrust value, and the other value is a corresponding speed. The following is an example mapping given in string form, and rendered in Figure 53.

```
thrust_map = "-100:-3.5, -75:-3.2, -10:-2, 20:2.4, 50:4.2, 80:4.8, 100:5"
```

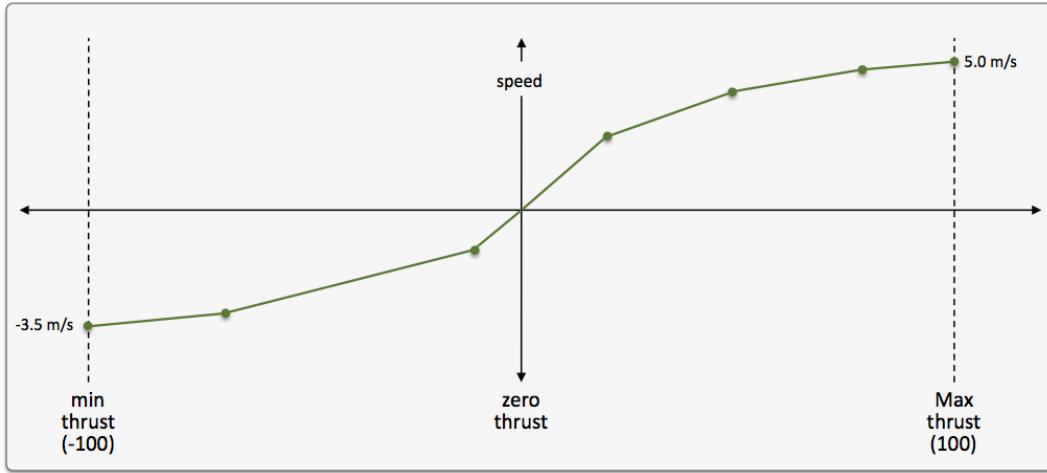


Figure 53: **A Thrust Map:** The example thrust map was defined by seven mapping points in the string ”-100:-3.5, -75:-3.2, -10:-2, 20:2.4, 50:4.2, 80:4.8, 100:5”.

9.7.1 Automatic Pruning of Invalid Configuration Pairs

The thrust map has an immutable domain of $[-100, 100]$, indicating 100% forward and reverse thrust. Mapping pairs given outside this domain will simply be ignored. The thrust mapping must also be monotonically increasing. This follows the intuition that more positive thrust will not result in the vehicle going slower, and likewise for negative thrust. Since the map is configured with a sequence of pairs as above, a pair that would result in a non-monotonic map is discarded. All maps are created as if they had the pair $0:0$ given explicitly. Any pair provided in configuration with zero as the thrust value will be ignored; zero thrust always means zero speed. Therefore, the following map configurations would all be equivalent to the map configuration above and shown in Figure 53:

```
thrust_map = -120:-5, -100:-3.5, -75:-3.2, -10:-2, 20:2.4, 50:4.2, 80:4.8, 100:5.0, 120:6
thrust_map = -100:-3.5, -75:-3.2, -10:-2, 20:2.4, 50:4.2, 80:4.8, 90:4, 100:5.0
thrust_map = -100:-3.5, -75:-3.2, -10:-2, 0:0, 20:2.4, 50:4.2, 80:4.8, 100:5.0
thrust_map = -100:-3.5, -75:-3.2, -10:-2, 0:1, 20:2.4, 50:4.2, 80:4.8, 100:5.0
```

In the first case, the pairs "-120:-5" and "120:6" would be ignored since they are outside the $[-100, 100]$ domain. In the second case, the pair "90:4" would be ignored since its inclusion would entail a non-monotonic mapping given the previous pair of "80:4.8". In the third case, the pair "0:0" would be effectively ignored since it is implied in all map configurations anyway. In the fourth case, the pair "0:1" would be ignored since a mapping from a non-zero speed to zero thrust is not permitted.

9.7.2 Automatic Inclusion of Implied Configuration Pairs

Since the domain $[-100, 100]$ is immutable, the thrust map is altered a bit automatically when or if the user provides a configuration without explicit mappings for the thrust values of -100 or 100 . In this case, the missing mapping becomes an implied mapping. The mapping $100:v$ is added where v is the speed value of the closest point. For example, the following two configurations are equivalent:

```
thrust_map = -75:-3.2, -10:-2, 20:2.4, 50:4.2, 80:4.8
thrust_map = -100:-3.2, -75:-3.2, -10:-2, 20:2.4, 50:4.2, 80:4.8, 100:4.8
```

9.7.3 A Shortcut for Specifying the Negative Thrust Mapping

For convenience, the mapping of positive thrust values to speed values can be used in reverse for negative thrust values. This is done by configuring `uSimMarine` with `thrust_reflect=true`, which is false by default. If `thrust_reflect` is false, then a speed of zero is mapped to all negative thrust values. If `thrust_reflect` is true, but the user nevertheless provides a mapping for a negative thrust in a thrust map, then the `thrust_reflect` directive is simply ignored and the thrust map is used instead. For example, the following two configurations are equivalent:

```
thrust_map = -100:-5, -80:-4.8, -50:-4.2, -20:-2.4, 20:2.4, 50:4.2, 80:4.8, 100:5
```

and

```
thrust_map = 20:2.4, 50:4.2, 80:4.8, 100:5
thrust_reflect = true
```

9.7.4 The Inverse Mapping - From Speed To Thrust

Since a thrust map only permits configurations resulting in a non-monotonic function, the inverse also holds (almost) as a valid mapping from speed to thrust. We say "almost" because there is ambiguity in cases where there is one or more plateau in the thrust map as in:

```
thrust_map = -75:-3.2, -10:-2, 20:2.4, 50:4.2, 80:4.8
```

In this case a speed of 4.8 maps to any thrust in the range $[80, 100]$. To remove such ambiguity, the thrust map, as implemented in a C++ class with methods, returns the lowest magnitude thrust in such cases. A speed of 4.8 (or 5 for that matter), would return a thrust value of 80. A speed of

-3.2 would return a thrust value of -75 . The motivation for this way of disambiguation is that if a thrust value of 80 and 100 , both result in the same speed, one would always choose the setting that conserves less energy. Reverse mappings are not used by the `uSimMarine` application, but may be of use in applications responsible for posting a desired thrust given a desired speed, as with the `pMarinePID` application.

9.7.5 Default Behavior of an Empty or Unspecified ThrustMap

If `uSimMarine` is configured without an explicit `thrust_map` or `thrust_reflect` configuration, the default behavior is governed as if the following two lines were actually included in the `uSimMarine` configuration block:

```
thrust_map      = 100:5
thrust_reflect = false
```

The default thrust map is rendered in Figure 54.

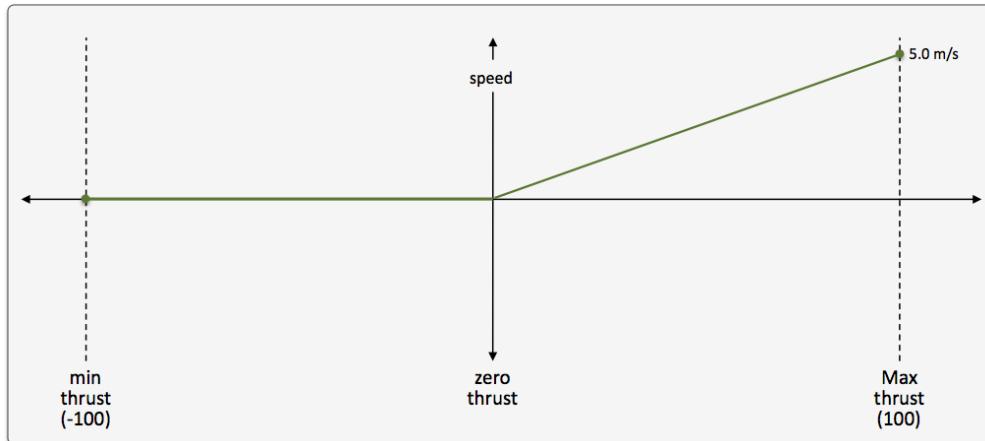


Figure 54: **The Default Thrust Map:** This thrust map is used if no explicit configuration is provided.

This default configuration was chosen for its reasonableness, and to be consistent with the behavior of prior versions of `uSimMarine` where the user did not have the ability to configure a thrust map.

10 The uMAC Utilities

In this section the utilities for viewing appcasts are discussed. The include:

- The `uMACView` utility: A GUI tool containing navigation tools for viewing appcasts across multiple vehicles or nodes. A snapshot is shown on the right in Figure 7.
- The `uMAC` utility: A utility implement in the terminal window. Simpler in the interface, but notable in that it allows a user to remotely launch the tool on deployed vehicle.
- `uMACView` integrated with `pMarineViewer`: An interface nearly identical with `uMACView` fully integrated within `pMarineViewer`, convenient for those already using `pMarineViewer`.

10.1 The uMACView Utility

The `uMACView` utility is an appcast viewer with a graphical user interface using the FLTK library. It contains three panes as shown in Figure 55. The bottom pane renders incoming appcasts. The top right pane lists possible applications for selecting appcast viewing for the present node. The upper left pane shows all presently known nodes from which appcasts have been received.

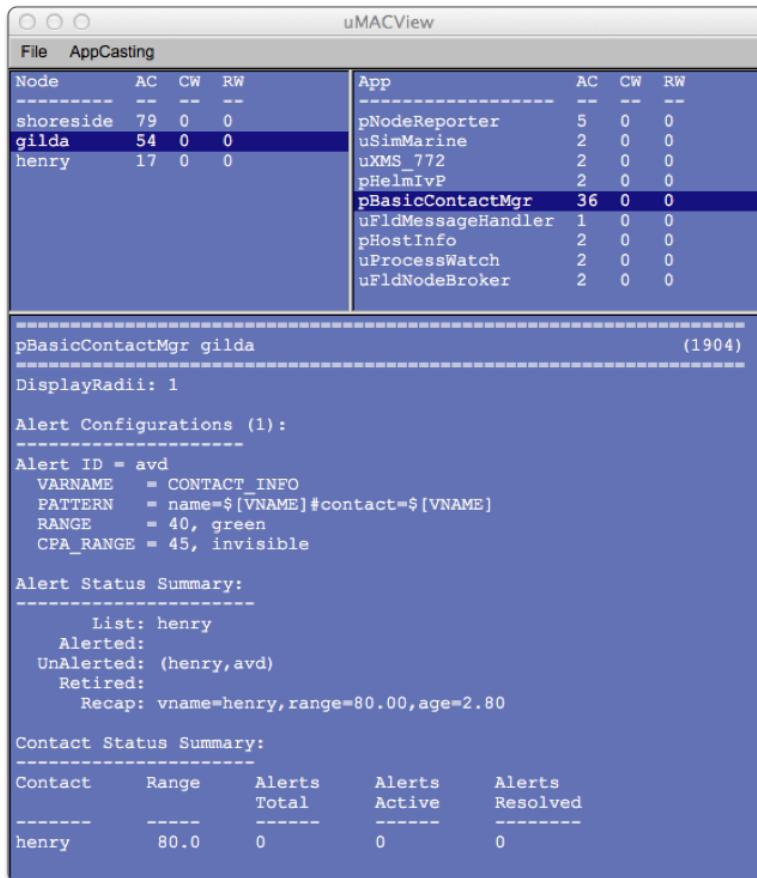


Figure 55: The uMACView utility receives appcasts from perhaps several different nodes and applications on each node. The interface allows the user to select a node and application for viewing the selected application's latest appcast. The viewer will raise alerts from non-selected nodes and applications when or if a run warning occurs.

The content of the *appcast pane* is solely determined by the appcast content itself. From the viewer's perspective it is simply pushing a list of strings to a browser pane. Even the formatting of the header lines showing the application name, node name and application iteration, are formatted by a method defined over the AppCast class.

The content of the *application pane* lists the applications known thus far to the viewer from incoming appcasts for a particular node. The order is shown by the order in which they were received. The three columns, "AC", "CW", and "RW", show the number of appcasts received from the application, and the number of run warnings and configuration warnings in the latest appcast.

The content of the *node pane* in the upper left lists the nodes known thus far to the viewer from incoming appcasts. The order is shown by the order in which they were received. The three columns, "AC", "CW", and "RW", show the number of appcasts received from a given node for *all* applications from that node, as well as the sum of all run warnings and configuration warnings for all applications received from the given node.

10.1.1 Publications and Subscriptions

The sole subscription for `uMACView` is the appcast message in the MOOS variable `APPCAST`. The sole publication is the appcast request, in the variable `APPCAST_REQ`.

10.1.2 Configuration File Parameters

A few configuration parameters are exposed to facilitate visual preference settings that would otherwise need to be done each time the application is launched via pull-down menus. These parameters are listed below, but may be recalled anytime by typing on the command line: "`uMACView -e`".

Listing 10.1: Configuration Parameters for uMACView.

<code>nodes_font_size</code> :	Possible settings, <code>xsmall</code> , <code>small</code> , <code>medium</code> , <code>large</code> . The default is <code>medium</code> .
<code>procs_font_size</code> :	Possible settings, <code>xsmall</code> , <code>small</code> , <code>medium</code> , <code>large</code> . The default is <code>medium</code> .
<code>appcast_font_size</code> :	Possible settings, <code>xsmall</code> , <code>small</code> , <code>medium</code> , <code>large</code> . The default is <code>small</code> .
<code>appcast_color_scheme</code> :	Possible settings, <code>default</code> , <code>beige</code> , <code>indigo</code> . The default is <code>default</code> .
<code>appcast_height</code> :	Possible settings, <code>[30,35,40,..., 85,90]</code> . The default is <code>70</code> .
<code>refresh_mode</code> :	Possible settings, <code>paused</code> , <code>events</code> , <code>streaming</code> . The default is <code>events</code> .

The `appcast_height` refers to the relative height of the appcast pane to the window. The default of 70 means the pane will be 70% of the overall window height.

The `refresh_mode` refers to the policy of refreshing appcasts via appcast requests sent to the nodes and their applications. This is discussed below in Section 10.1.4

10.1.3 Command Line Arguments and Options

A few command line arguments are available. They are similar to most other MOOS-IvP applications. These arguments are listed below, but may be recalled anytime by typing on the command line:

```
$ MACView --help or -h
```

- **--alias=<ProcessName>**: Launch with the given process name rather than uMACView.
- **--example, -e**: Display example MOOS configuration block.
- **--help, -h**: Display command line usage.
- **--interface, -i**: Display MOOS publications and subscriptions.
- **--version, -v**: Display release version information.

10.1.4 Refresh Modes

The **uMACView** utility, operates in one of three *refresh modes*. This mode is always shown on the upper right in the title bar in reverse color. The three modes are the *streaming*, *events*, and *paused* modes.

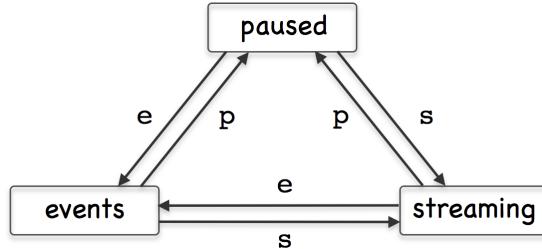


Figure 56: The uMAC utility is in one of three *refresh modes*, determining the manner in which appcast requests are conveyed to known applications and nodes. Reserved keyboard keys are used to transition between modes.

The Paused Refresh Mode In the *paused* refresh mode, no appcasts are solicited by the **uMAC** utility. The information being rendered should remain constant, virtually paused. Since appcast requests have a duration associated with them, prior appcast requests received by an application may need some time before expiring. Therefore the **uMAC** user may still see a trickle of updates after entering the paused mode. Furthermore, if there is another **uMAC** utility open, generating appcast requests, updates might still be seen after pausing.

The Events Refresh Mode In the *events* refresh mode, appcast requests of two types are sent to all known nodes and applications. The first type of request is sent only to the selected node and application. This type requests a continual update of appcasts, unconditionally. This application is, after all, the selected application for viewing. The second type of request is sent to all other nodes and applications requesting an appcast *only if a new run warning is generated*. The *events* refresh mode is the default mode upon launch.

The Streaming Refresh Mode In the *streaming* refresh mode, appcast requests are sent to all nodes and all applications, all the time. Furthermore, the request type is unconditional, meaning the application is requested to post a new appcast regardless of whether anything has changed in the application status. This mode is normally used for brief debugging, perhaps to check whether a node or application fails to respond. It creates a lot of appcast messaging traffic. It is not recommended to operate in this mode normally.

10.2 The uMAC Utility

The **uMAC** utility is an appcast viewer implemented in the terminal console. It acts the same as **uMACView** in terms of soliciting appcasts by publishing **APPCAST_REQ** messages and receiving **APPCAST** mail. The advantage over the GUI tool is the ability to remotely log into a vehicle and launch **uMAC** in a terminal window. This is especially useful if the vehicle is otherwise not behaving in its communication with a shoreside MOOS community. Like **uMACView**, multiple versions of the utility may be running at the same time without interference with one another.

10.2.1 Content Modes

The viewable information in the **uMAC** tool is rendered in one of four *content modes*. Besides a *help* mode, the other three modes correlate to one of the three panes of the **uMACView** window in Figure 55. The primary content mode is the *appcast content mode*, where the output is an appcast of a particular application as shown in Figure 57.

```

uMAC_4430:  Nodes (3)                                     (6) EVENTS
pBasicContactMgr gilda                                    (93)
DisplayRadii: 1

Alert Configurations (1):
-----
Alert ID = avd
  VARNAME   = CONTACT_INFO
  PATTERN   = name=${VNAME}#contact=${VNAME}
  RANGE     = 40, green
  CPA_RANGE = 45, invisible

Alert Status Summary:
-----
      List: henry
      Alerted:
      UnAlerted: (henry,avd)
      Retired:
      Recap: vname=henry,range=80.00,age=1.24

Contact Status Summary:
-----
Contact    Range    Alerts    Alerts    Alerts
          Total    Active    Resolved
-----
henry      80.0     0        0        0

```

Figure 57: The **uMAC** utility monitors appcasts from a terminal window. The user may switch between appcasting sources by navigating with a keyboard menu. The primary advantage of **uMAC** is the ability to run it on a remotely deployed vehicle with a network connection.

The content of this window should look very similar to the bottom pane of the `uMACView` window in Figure 55. Two other content modes are supported, the *nodes content mode*, and the *procs content mode*. The former allows the user to select between different nodes, i.e., vehicles. The latter allows selection between different processes (MOOS applications) for the selected node. These modes correlate to the top two panes in the `uMACView` tool shown in Figure 55. A fourth, help, content mode is also supported. Transitioning between modes usually is done by selecting one of the choices presented, or popping back up a mode to allow a higher level choice. This done with three reserved keyboard keys, `p`, `n`, `h`, implementing the transitions shown in Figure 58.

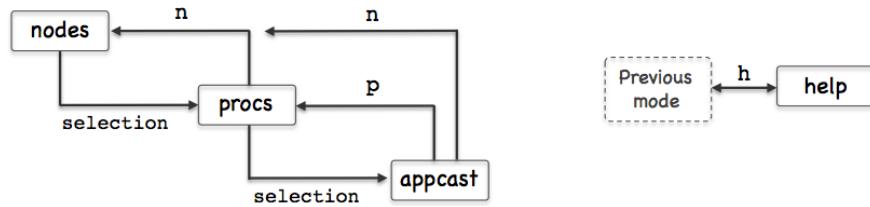


Figure 58: The `uMAC` utility monitors appcasts from a terminal window. The

An example rendering of `uMAC` in the *nodes content mode* is shown on the left in Figure 59. Each discovered node is assigned a character ID in the left-most column for selecting the node. The ID's are assigned as the nodes are discovered from incoming `APPCAST` messages.

The title line at the top of the report shows, on the left, the name of the `uMAC` application as it is known to the MOOSDB, and the number of nodes discovered. When `uMAC` is launched it gives itself a random suffix, such as `uMAC_5991` in this example, to allow multiple `uMAC` sessions connect with the same MOOSDB. Recall the MOOSDB requires unique names across applications. The righthand side of the title line shows the number of iterations of the `uMAC` in parentheses, and the *refresh mode* shown in reverse color.

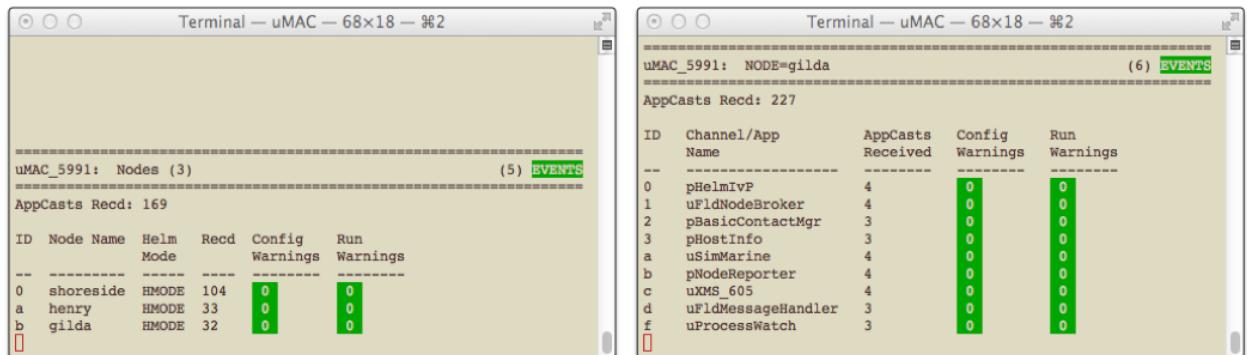


Figure 59: The `uMAC` utility monitors appcasts from a terminal window. The user may switch between appcasting sources by navigating with a keyboard menu. The primary advantage of `uMAC` is the ability to run it on a remotely deployed vehicle with a network connection.

An example from the *procs content mode* is shown on the right in Figure 59. Each discovered process (MOOS application) is assigned an ID in the left-most column for selecting the process.

The ID's are assigned as the apps are discovered from incoming APPCAST messages. The title line at the top is nearly the same format as in the *nodes* content mode, except that the selected node is shown rather than the total nodes. The appcast counter just below the title line indicates the total number of appcasts received over all nodes, not just the selected node.

10.2.2 Refresh Modes

The uMAC utility, like the uMacView utility, operates in one of three *refresh modes*. This mode is always shown on the upper right in the title bar in reverse color. The three modes are the *streaming*, *events*, and *paused* modes. The description of these modes, given in Section 10.1.4, is also applicable to the operation for the uMAC utility.

10.2.3 A Tip Regarding Process Monitoring and uMAC Sessions

The uProcessWatch application is a utility for monitoring the presence of applications connected to the MOOSDB. It is appcast enabled, and will post a run warning when it detects the disappearance of a prior noted process. If a uMAC is launched and then exited, the uProcessWatch utility may interpret this as a problem and post a run warning. The effect may be that real run warnings are then later ignored. Tip: Add the following configuration line to uProcessWatch to ignore the exit of a uMAC session: `nowatcth = uMAC*`.

10.2.4 Publications and Subscriptions

The sole subscription for uMacView is the appcast message in the MOOS variable APPCAST. The sole publication is the appcast request, in the variable APPCAST_REQ.

10.2.5 Configuration File Parameters

There are no configuration file parameters specific to uMAC.

10.2.6 Command Line Arguments and Options

A few command line arguments are available. They are similar to most other MOOS-IvP applications. These arguments are listed below, but may be recalled anytime by typing on the command line:

```
$ uMAC --help or -h
```

- **--alias=<ProcessName>**: Launch with the given process name rather than uMACView.
- **--example, -e**: Display example MOOS configuration block.
- **--help, -h**: Display command line usage.
- **--interface, -i**: Display MOOS publications and subscriptions.
- **--version, -v**: Display release version information.

10.3 The uMACView Utility Integrated with pMarineViewer

The final uMAC tool is essentially `uMACView` embedded in the `pMarineViewer` application as shown in Figure 60. This is mostly just a convenience for users already using `pMarineViewer`. The appcasting mode may be toggled with the '`a`' key to return to the traditional viewing layout. The `AppCasting` pull-down menu offers the same set of selections as `uMACView` with the exception of the hot keys used to switch between appcasting refresh modes since those keys were already used for other things in `pMarineViewer`.

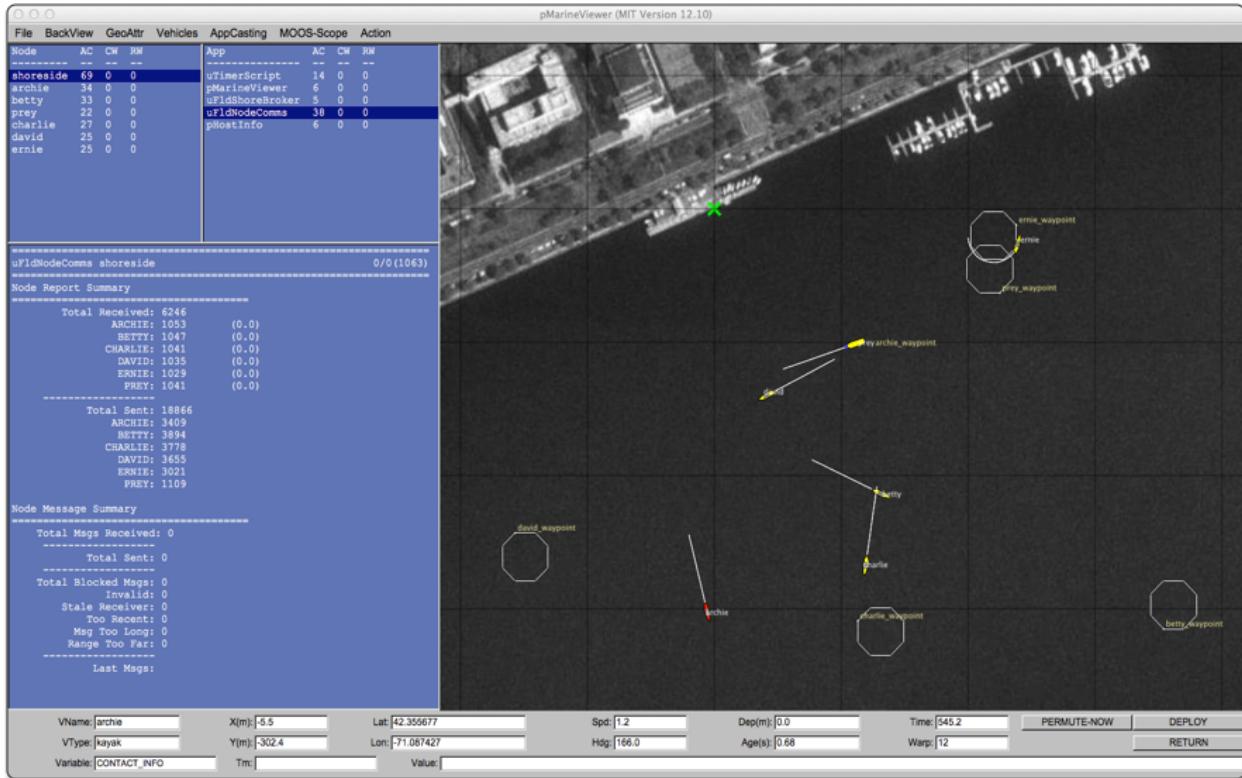


Figure 60: The `pMarineViewer` utility has an appcasting viewing capability very similar to `uMACView` embedded in the viewer. The rendering of the appcasting panes may be toggled on/off with the '`a`' key.

In addition to toggling on/off the appcasting portion of the window, the width of the set of appcasting panes may be made wider or thinner using the `CTRL-ALT-ARROW` keys. By default the appcasting panes consume 30% of the width. The default height of the appcasting (bottom) portion of the appcasting panes consume 75% of the height of the set of appcasting panes. These startup extents may be changed with configuration the parameters:

```
appcasting_width = 25    // legal values [20, 25, ..., 65, 70]
appcasting_height = 80   // legal values [30, 35, ..., 85, 90]
```

The prevailing value of these parameters can always be discovered by checking which radio button in the pull-down menu is presently selected.

11 Enabling a MOOS Application for AppCasting

In this section we discuss the steps for enabling a new or existing MOOS application to support appcasting. Much of the requisite appcasting source code is the same for any application. This is captured in a new `AppCastingMOOSApp` class to minimize appcasting boilerplate code. This class, as indicated in Figure 61, is a subclass of the common `CMOOSApp` class distributed with core MOOS.

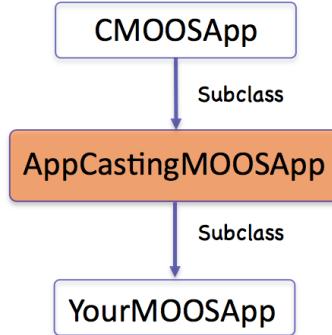


Figure 61: On-demand appcasting is implemented in a new `CMOOSApp` subclass called `AppCastingMOOSApp`.

The following is the complete list of required steps:

- Subclass the `AppCastingMOOSApp` superclass,
- Invoke a pair of superclass methods in the `Iterate()` function,
- Invoke a superclass method in the `OnNewMail()` function,
- Invoke a superclass method in the `OnStartUp()` function,
- Invoke a superclass method when registering for variables,
- Implement a `buildReport()` function where appcasts are formed.

In our discussions to follow, a hypothetical `YourMOOSApp` application and class definition is described. The above steps are the minimum requirements for appcasting, and they are fairly boilerplate, in most cases a single line of code. To make good use of the appcasting features, configuration warnings, run warnings and events may be placed in your application code. This is neither mandatory nor boilerplate, but really dependent on the application. Nevertheless, a few rules of thumb discussed:

- Posting events,
- Posting run warnings,
- Posting configuration warnings.

[2]

11.1 Sub-classing the `AppCastingMOOSApp` Superclass

The first step is to make `YourMOOSApp` a subclass of the `AppCastingMOOSApp`. This brings your application class everything from the traditional `CMOOSApp` class as well as the appcasting features

of the `AppCastingMOOSApp` class. The only additional thing besides declaring the superclass is to declare the `buildReport()` function. This virtual function is invoked when an appcast has been deemed warranted. Appcasts are not typically generated on each iteration. See the later discussion about *on-demand appcasting*. The contents of the `buildReport()` function are discussed in greater detail in Section 11.6.

Listing 11.1: Pseudocode for sub-classing the `AppCastingMOOSApp` superclass.

```

1 #include "MOOS/libMOOS/Thirdparty/AppCasting/AppCastingMOOSApp.h"
2
3 class YourMOOSApp : public AppCastingMOOSApp           // Instead of CMOOSApp
4 {
5     // All your normal class declaration stuff
6
7     bool buildReport();                                // Add this line
8 };

```

11.2 Invoking Superclass Methods in the `Iterate()` Method

The next step is to implement `YourMOOSApp::Iterate()` to invoke two superclass functions; one at the very beginning and one at the very end. The first superclass function, on line 2 below, does certain common bookkeeping such as incrementing the counter representing the number of application iterations, and updating a variable holding the present MOOS time. The second superclass function, on line 6, invokes the on-demand appcasting logic discussed previously. If an appcast is deemed warranted, it will invoke the `buildReport()` function.

Listing 11.2: Pseudocode for invoking subclass methods in the `Iterate()` method.

```

1 bool YourMOOSApp::Iterate()
2 {
3     AppCastingMOOSApp::Iterate();                      // Add this line
4
5     // Do all your normal Iterate stuff
6
7     AppCastingMOOSApp::PostReport();                  // Add this line
8     return(true);
9 }

```

11.3 Invoking a Superclass Method in the `OnNewMail()` Method

The next step is to implement `YourMOOSApp::OnNewMail()` to invoke a superclass function to have the first opportunity to handle incoming mail. For example, `APPCAST_REQ` mail is handled in the superclass. The list of mail messages is passed by reference to the superclass handler, allowing the `AppCastingMOOSApp::OnNewMail()` function to remove handled messages before returning to the mail handling implemented in `YourMOOSApp::OnNewMail()`.

Listing 11.3: Pseudocode for invoking a superclass method in the `OnNewMail()` method.

```

1 bool YourMOOSApp::OnNewMail(MOOSMSG_LIST &NewMail)
2 {
3     AppCastingMOOSApp::OnNewMail(NewMail);           // Add this line
4
5     // Do all your other normal mail handling.
6 }
```

11.4 Invoking a Superclass Method in the OnStartUp() Method

The next step is to implement `YourMOOSApp::OnStartUp()` to invoke a superclass function to perform startup steps needed by the `AppCastingMOOSApp` superclass.

Listing 11.4: Pseudocode for invoking a superclass method in the `OnStartUp()` method.

```

1 void YourMOOSApp::OnStartUp()
2 {
3     AppCastingMOOSApp::OnStartUp();                  // Add this line
4
5     // Do all your other startup stuff
6 }
```

11.5 Invoking a Superclass Method When Registering for Variables

The next step is to invoke the `AppCastingMOOSApp::RegisterVariables()` wherever variables are registered in `YourMOOSApp` implementation. Many application developers have, in practice, created a dedicated `registerVariables()` function, typically invoked at the conclusion of both `OnStartUp()` and `OnConnectToServer()`. The following example is one way to handle this.

Listing 11.5: Pseudocode for invoking a superclass method when registering for variables.

```

1 void YourMOOSApp::registerVariables()
2 {
3     AppCastingMOOSApp::RegisterVariables();          // Add this line
4
5     // Do all your other registrations
6 }
```

11.6 Implementing a buildReport Method for Generating AppCasts

The `buildReport()` function is where appcasts are made! The action that happens here is unique to the application. The form is designed by the application developer to reflect the most meaningful, concise snapshot of the application's present status. Recall that it is invoked automatically when or if the application deems an appcast is to be generated. Those issues were discussed in Section 11.10, and 11.2. For the purposes here though, a decision has indeed been made to generate an appcast, and `buildReport()` has been invoked to see that it happens. A simple example of `buildReport()` is shown below in Listing 6

Listing 11.6: Pseudocode for a very simple `buildReport()` example.

```

1 bool YourMOOSApp::buildReport()
2 {
3     m_msgs << "Total Message Report: \n";
4     m_msgs << "# of good messages: " << m_good_message_count << endl;
5     m_msgs << "# of bad messages: " << m_bad_message_count << endl;
6
7     return(true);
8 }
```

This simple example would generate something similar to the appcast rendered in Figure 62.

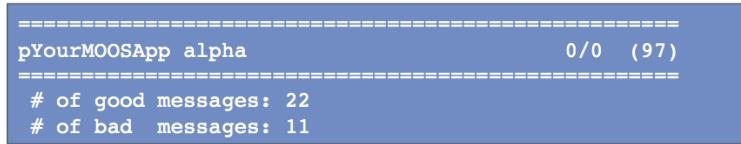


Figure 62: The rendering of a very simple appcast with just two message lines, no warnings, and no events. The header bar shows the name of the application, the originating MOOS community, the number of configuration and run warnings, and the application's current iteration counter.

Assume for the sake of the example that the two counter variables at the end of lines 2 and 3 are member variables for the fictitious `YourMOOSApp` class. The member variable `m_msgs` however is an STL stringstream also declared in the `AppCastingMOOSApp` class, for holding message output. The primary means of building an appcast is to add successive lines to the appcast via:

```
m_msgs << <element> << endl;
```

where `<element>` may be a `string`, `double`, `int`, `unsigned int`, or any combination joined by the "`<<`" operator. A new line is indicated by tacking on the newline, "`\n`", at the end. That's pretty much it, but there are a few other noteworthy points:

- Run warnings, configuration warnings, and events are not added during `buildReport()`, though not strictly prevented. They are more typically added as warnings are discovered, or events occur during the normal mail handling or iterate cycle.
- The header lines shown in Figure 62 is made automatically by the uMAC tool. They grab information in the appcast such as the application name and iteration number. These are filled by the boilerplate function calls such as `AppCastingMOOSApp::Iterate()` described earlier.
- The appcast is automatically cleared prior to each invocation of `buildReport()`. Warnings and events however are not cleared as discussed earlier.
- Returning true indicates that the appcast was indeed populated. Returning false would result in the appcast not being sent to the terminal or published to the MOOSDB. This is useful for some applications that may want to apply additional criteria before deciding to appcast.
- Although report formatting, e.g., columns or tables, is not natively supported somehow in the `buildReport()` routine, there are tools available that facilitate formatting that work easily with the `buildReport()` interface. They are discussed later in Section 99.

11.7 Posting Events

Events are messages (strings) that the application developer deems to be noteworthy enough to want to include in appcast output. An event may be posted anywhere in the application code with the `reportEvent()` function defined in the `AppCastingMOOSApp` class. A simple example:

```
reportEvent("Good msg received: " + message);
```

When the appcast is rendered in either a uMAC tool or in the terminal, the result would look similar to that in Figure 63.

```
=====
pYourMOOSApp alpha          0/1 (97)
=====
RunTime Warnings: 14
[11] Bad msg received: bricks

# of good messages: 1
# of bad messages: 14

=====
Most Recent Events (22):
=====
[23.09] Good msg received: happyjoy
```

Figure 63: The rendering of a simple appcast with two message lines, a single run warning, and single event. The header bar shows the name of the application, the originating MOOS community, the number of configuration and run warnings, and the application's current iteration counter.

Recall that only a limited number of events are retained. Older events are dropped once a maximum amount is exceeded. The default event list size is eight, but this may be overridden for a particular application with the following parameter setting in the applications MOOS configuration block:

```
max_appcast_events = 25
```

The event list size may be set to at most 100.

11.8 Posting Run Warnings

Run warnings are similar to events, but they convey that something may have gone wrong. A run warning may be posted anywhere in the application code with the `reportRunWarning()` function defined in the `AppCastingMOOSApp` class. A simple example:

```
reportRunWarning("Bad msg received: " + message);
```

The appcast structure and uMAC tools are implemented such that run warnings are more easily brought to the attention of the operator. This is due to the following reasons:

- When an appcast has a run warning, the uMAC utilities will indicate so by turning the text red, as in Figure 6. Even when the focus of the uMAC utility is not on the particular appcast containing the run warning, the menu items for the appcast and vehicle will be rendered red. In Figure 64 for example, the menu browser focus is on vehicle `archie` and the `uFldMessageHandler` application. The red highlights also indicate there is a run warning on another application on `archie`, and there is also a run warning on vehicle `charlie`.
- An appcast request to an application may specify the reporting threshold to be "run_warning" as discussed in Section 11.10.4. In this case an application will repeatedly choose not to publish an appcast unless a new run warning has been generated.
- The uMAC tools also keep a running tally of run warnings for each vehicle and each application under the column labeled "RW" as shown in Figure 64.

File AppCasting			
Node	AC	CW	RW
<code>archie</code>	200	0	1
<code>shoreside</code>	25	0	0
<code>betty</code>	26	0	0
<code>david</code>	28	0	0
<code>prey</code>	21	0	0
<code>charlie</code>	23	0	1
<code>ernie</code>	28	0	0

App	AC	CW	RW
<code>pNodeReporter</code>	97	0	0
<code>pBasicContactMgr</code>	7	0	0
<code>uFldMessageHandler</code>	82	0	0
<code>pHelmIpvP</code>	3	0	0
<code>uProcessWatch</code>	3	0	1
<code>pHostInfo</code>	4	0	0
<code>uFldNodeBroker</code>	4	0	0

Figure 64: The uMAC tools include will highlight an application that has produced an appcast with a run warning. If the run warning occurred on a node not currently in focus, e.g., `charlie` in the figure, the node itself is highlighted. The user can then select the other node to view the application generating the run warning.

Recall that only a limited number of run warnings are retained. Unlike events where the older ones are dropped once a maximum has been exceeded, old run warnings are never dropped. After the maximum has been reached, the generic warning "Other Run Warnings" is simply incremented. The default list size is ten, but this may be overridden for a particular application with the following parameter setting in the applications MOOS configuration block:

```
max_appcast_run_warnings = 50
```

The run warning list size may be set to at most 100.

11.9 Posting Configuration Warnings

Configuration warnings are similar to run warnings but they are typically only posted during the application startup, when the mission configuration file is read. A configuration warning may be posted with the `reportConfigWarning()` function defined in the `AppCastingMOOSApp` class. A simple example:

```
reportConfigWarning("Problem configuring FOOBAR. Expected a number but got: " + str);
```

There is a second way to post a configuration warning. This second method takes as an argument the original full configuration parameter line found in the mission file. Before posting the configuration warning it checks to see if the parameter was something that likely was handled by the superclass. This prevents the application from reporting that `AppTick=10"` is an unknown parameter for example.

```
reportUnhandledConfigWarning(original_full_config_line);
```

The appcast structure and uMAC tools are implemented such that configuration warnings are more easily brought to the attention of the operator. This is due to the following reasons:

- When an appcast has a configuration warning, the uMAC utilities will indicate so by turning the text green, as in Figure 65. Even when the focus of the uMAC utility is not on the particular appcast containing the config warning, the menu items for the appcast and vehicle will be rendered green. In Figure 65 for example, the menu browser focus is on the `shoreside` and the `uTimerScript` application application. The green highlights indicate there is a configuration warning in the `uFldShoreBroker` application and on other applications on `archie`, `charlie`, and `ernie`.
- The uMAC tools also keep a running tally of configuration warnings for each vehicle and each application under the column labeled "CW" as shown in Figure 65.

File AppCasting			
Node	AC	CW	RW
<code>shoreside</code>	102	1	0
<code>david</code>	16	0	0
<code>prey</code>	12	0	0
<code>betty</code>	16	0	0
<code>archie</code>	16	2	0
<code>charlie</code>	16	1	0
<code>ernie</code>	16	1	0

App	AC	CW	RW
<code>uTimerScript</code>	91	0	0
<code>uMACView</code>	3	0	0
<code>uFldNodeComms</code>	2	0	0
<code>pMarineViewer</code>	2	0	0
<code>pHostInfo</code>	2	0	0
<code>uFldShoreBroker</code>	2	1	0

Figure 65: The uMAC tools include will highlight an application that has produced an appcast with a configuration warning. If the configuration warning occurred on a node not currently in focus, e.g., `archie`, `charlie`, or `ernie` in the figure, the node itself is highlighted. The user can then select the other node to view the application generating the warning.

Like run warnings and events, configuration warnings are limited in number to prevent runaway growth in the size of an appcast over time. The limit however is large, 100, and fixed. Presumably the number of configuration warnings is limited by the number of possible configuration parameters for an application, and large number of configuration warnings usually indicates that a mission should be halted and fixed before moving on.

The example code in Listing 7 below is an example `OnStartUp()` method showing the intended scenarios of reporting configuration warnings.

Listing 11.7: Pseudocode example for `OnStartUp()` configuration warning handling.

```

1 bool YourMOOSApp::OnStartUp()
2 {
3     AppCastingMOOSApp::OnStartUp();
4
5     STRING_LIST sParams;
6     if(!m_MissionReader.GetConfiguration(GetAppName(), sParams))
7         reportConfigWarning("No config block found for " + GetAppName());
8
9     STRING_LIST::iterator p;
10    for(p=sParams.begin(); p!=sParams.end(); p++) {
11        string orig  = *p;
12        string line  = *p;
13        string param = toupper(MOOSChomp(line, "="));
14        string value = line;
15
16        if(param == "FOO") {
17            bool handled = handleConfigFOO(value);
18            if(!handled)
19                reportConfigWarning("Problem with configuring FOO: " + value);
20        }
21        else if(param == "BAR")
22            bool handled = handleConfigBAR(value);
23            if(!handled)
24                reportConfigWarning("Problem with configuring BAR: " + value);
25    }
26
27    else
28        reportUnhandledConfigWarning(orig);
29    }
30    return(true);
31 }

```

There are a few issues worth noting in this example:

- A check is made that the application actually has a configuration block. This is done on lines 5-6, and catches a common bug with newly minted mission files.
- Checks are made that known parameters have legal values. This is done for the parameters `FOO` in lines 15-19 and `BAR` in lines 20-24 in Listing 6. In each case an external handler is invoked, e.g., `handleConfigFOO(value)` on line 16, which returns a Boolean indicating whether the parameter value was proper or not. If not, a configuration warning is reported as on lines 18 and 23.
- A final case is handled (lines 26-27) if the present parameter is not matched by any of the previous cases. This catches the common mistake of mis-spelling the parameter name. The `reportUnhandledConfigWarning()` function is used rather than the `reportConfigWarning()` function. The former function takes the whole original configuration line as input and checks to see if the parameter was a parameter handled at the superclass level. This prevents the generation of a warning for a line like `AppTick=5`.

11.10 Under The Hood - On-Demand AppCasting

On-demand appcasting refers to the goal of minimizing generated appcasts, ideally only when there is a reasonable chance that an appcast will be tended to (looked at) by a user. Users may be tending to an appcast either by looking at terminal output or through a uMAC utility.

11.10.1 Motivation

Consider the following scenario:

- 50 simulated vehicles,
- 10 MOOS applications on each vehicle,
- Each application running with an apptick of 4Hz,
- MOOS time warp running at 50x real time.

In the above scenario (a conceivable scenario in our lab), 100,000 appcasts would be generated *per second*. Consider a second scenario:

- One fielded underwater vehicle,
- 10 MOOS applications,
- Each application running with an apptick of 4Hz,
- Mission duration 6 months.

Although only 40 appcasts per second are generated, the vehicle is underwater and, limited to acoustic messages, likely no appcast will ever be viewed. With a six month mission, the CPU time and power budget needed to generate those 40 reports per second may come under scrutiny.

11.10.2 AppCast Generation Criteria

An appcast will be generated on any given iteration only if the contingencies and criteria depicted in Figure 66 are met. In short, an appcast is generated if either a terminal is open or a uMAC tool is requesting the appcast. Even when appcasts are being generated, they may be generated less frequently than each iteration, to be more in line with the frequency a user is able to process refreshed report information. These issues are discussed next.

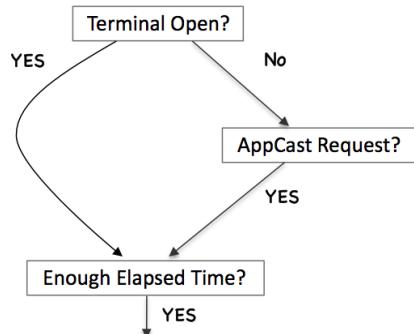


Figure 66: The appcast-generation decision is based on a few checks and contingencies. An appcast is generated only when tended to by a user, and only as often as a user may reasonably expect to process updated reports.

11.10.3 Terminal Switching

In the flow diagram of Figure 66, the first step in determining whether an appcast is to be generated involves the presence of a terminal. If an app is launched with a terminal window open, or launched within a terminal window, appcasts should be generated. In this regard, appcasting apps should behave like non-appcasting apps, although the former produce its output by different means. From the user's perspective, launching an application with or within a terminal window should result in the same familiar behavior regardless of the application.

Upon startup an AppCastingMOOSApp application will try to determine if information directed to `stdout` will make its way to an open terminal window. The following reasoning is applied:

- By default the application assumes information directed to `stdout` is indeed being rendered in a terminal window.
- During startup, when mission file parameters are examined, if the application detects the presence of a global parameter, `TERM_REPORTING`, and it is set to "false", then the application assumes that a terminal window is not open to receive output written to `stdout`.
- During application startup, the following library utility function is invoked:
`int isatty(int);`

This function is defined in "`unistd.h`" and is able to detect whether `stdout` is receiving output.

[2]

The above ensures that the application will err on the side of always producing appcasts/output to the terminal. To ensure otherwise, make sure to include `TERM_REPORTING="false"` in the MOOS configuration file. The last check is just a convenience or extra fail-safe; if an application is launched with `pAntler` using `NewConsole=false` and `pAntler` itself is also launched with `stdout` redirected to `/dev/null/`, then the application will automatically detect this. This style of launching is actually a fairly common scenario in launching MOOS communities on vehicles in our lab. In detecting this situation, the application will not generate an appcast unless a uMAC tool is explicitly requesting it. This is the next step in the flow diagram of Figure 66, and discussed next.

11.10.4 AppCast Requests

An *appcast request* is message sent to an application to begin or continue generating appcasts for some specified period of time. The request may originate in the local MOOSDB community, or in a remote MOOSDB community as the two cases suggest in Figure 67.

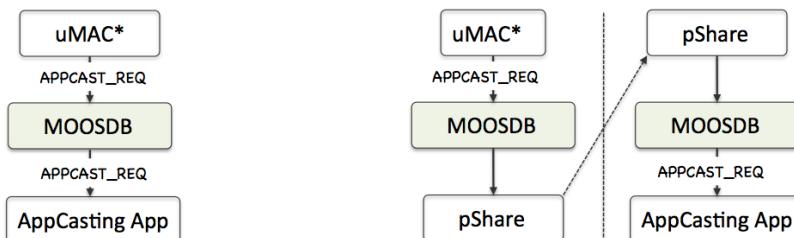


Figure 67: An appcast request requests that the receiving MOOS application begin or continue to generate appcasts for some specified period of time. The request may come from an app within the same MOOS community, or from an external community.

The content of an appcast request has the following fields:

- **node**: This must match the name of the MOOS community within which the application is running, otherwise the appcast request is ignored. If the `node` is "any", the match is always granted.
- **app**: This must match the name of the MOOS application receiving the appcast request, otherwise the request is ignored. If the `app` is "any", the match is always granted.
- **duration**: This number is given in seconds and represents the duration the appcast request would honored after time of receipt. The maximum value is 30.
- **threshold**: The threshold name indicates under what conditions the receiving application should generate an appcast. The two possible values are "any" and "run_warning". Their meaning is discussed below.
- **key**: The key helps the receiving application discern appcast requests from different applications.

An example `APPCAST_REQ` message:

```
APPCAST_REQ = "node=henry,app=uProcessWatch,duration=3.0,key=pMarineViewer:shore,thresh=any"
```

Node and Application Name Matching

An appcast request will be ignored by a receiving application if the request does not *match*. The match must be made between both (a) the requested and actual node name, and (b) the requested and actual application name. A requested value "any" will match to anything. In most circumstances the requesting application, e.g., `uMAC`, `uMACView` or `pMarineViewer`, will publish requests naming nodes and applications explicitly. Upon startup however, requests may be sent broadly to all nodes and all applications just to learn about existing appcasting sources before beginning to make requests explicitly.

Duration Time

An appcast request comes with a duration. If the requesting application disconnects, the duration caveat ensures the original request will timeout before too long, avoiding the perpetual generation of now unwanted appcasts. Typically if a uMAC tool is monitoring the appcasts of a particular application, it repeatedly sends an appcast request to that application, each time refreshing the timeout criteria.

Request Threshold

The appcast request will specify one of two criteria to the application. First, if the criteria is "any", the application is to publish an appcast on each possible occasion. This may not be the same as each iteration, since a further minimum reporting interval may be applied, as described next in Section 11.10.5. The second threshold type is "run_warning". This indicates to the receiving application that it should only generate an appcast when a new run warning has been added since the last appcast. Typically a uMAC tool will run in a mode sending appcast requests with the

latter threshold to virtually all nodes and apps, but appcast requests using the former threshold to a single node and app chosen by the user.

Request Key

Appcast requests specify a particular key, presumably a string that is unique to the originator. This allows the receiving application to do separate bookkeeping for each requesting application. As long as the appcast request threshold matches, hasn't timed out, and met the threshold criteria for *one* of its logged keys, then it indeed meets the criteria.

11.10.5 Limiting the AppCast Frequency

A third criteria is applied to the decision of generating an appcast on any given iteration. This criteria is depicted at the bottom of Figure 66. Even if a terminal window is open, or a valid appcast request has been received recently enough, the generation of an appcast may still be skipped if the previously generated appcast was generated too recently. Appcast content is meant to be human-readable, and humans can only read so fast. There is no sense in updating a terminal report say 50 times per second. Although most MOOS apps are typically not configured with an apptick of 50Hz, an apptick of 5 is fairly common, as well as a `MOOSTimeWarp` of say 10 or greater. By default, appcasting applications are limited in real-time frequency to once every 0.6 seconds. This number just reflects a number that, from experience, feels right for an update frequency. This can be overridden for any application with the following parameter in its configuration block:

```
term_report_interval = N
```

where N ranges from zero (appcast generation only limited by the iteration frequency) to at most 30 seconds.

11.10.6 Generating and AppCast vs. Publishing and AppCast

The flow chart shown in Figure 66 addresses the issue of whether or not an appcast is *generated*. Whether or not the appcast is *published* is a separate issue. Simply put, if a terminal window is open for the application, and it has not received any appcast requests, the appcast is generated (and rendered to the terminal `stdout`) but not published.

Below is informal logic shorthand for policy and conditions described over the last few pages. First, on the policy of whether an appcast is generated:

```
generate_appcast = (terminal || appcast_requested) && !recent_appcast
```

Next, on the question of whether an appcast is published:

```
publish_appcast = generate_appcast && appcast_requested
```

Both of the above depend on the term `appcast_requested` from the following expression:

```
appcast_requested = unexpired_request && ((request_threshold == "any") || new_run_warning)
```

The terms in this last expression will hopefully be apparent from the preceding pages.

11.10.7 Monitoring AppCast Traffic Volume

The uMAC tools provide a means for verifying that on-demand appcasting is behaving. These same tools are also useful for catching other anomalies. The information in Figure 68 below focuses on the information at the top of the [uMACView](#) tool depicted in Figure 7.

Node	AC	CW	RW
shoreside	79	0	0
ernie	16	0	0
david	44	0	0
archie	16	0	0
charlie	16	0	0
betty	16	0	0
prey	12	0	0

App	AC	CW	RW
uFldMessageHandler	4	0	0
uSimMarine	2	0	0
pHelmIpvP	2	0	0
pNodeReporter	2	0	0
uProcessWatch	28	0	0
pBasicContactMgr	2	0	0
uFldNodeBroker	2	0	0
pHostInfo	2	0	0

Figure 68: The uMAC tools include tallies of the number of appcasts received for each node (vehicle) and each application. The above is from the top of the [uMACView](#) and appcast tallies are shown under the "AC" column.

From the above figure, one might ask why so many appcasts have been received when the user is only focusing on one vehicle and one application. There are few answers to this question and each, listed below, are related to the need for a uMAC utility to *find* existing sources of appcasts. After all, how can it send an appcast request to a particular vehicle and particular application if it doesn't know that it exists?

- When an application starts up, it generates appcasts for the first few iterations. Even if no one is tending to this information, a few iterations worth of un-tended appcasts is not harmful. In practice this helps uMAC tools discover appcasting apps.
- When a uMAC tool starts up, it posts an appcast request to all known nodes and all known applications. The uMAC tool doesn't need to know or keep track of vehicles, but just simply posts `APPCAST_REQ="node=all,app=all, ..."` to its local MOOSDB. Other applications have the responsibility of bridging this variable to other vehicles as they are discovered.
- Each time a uMAC tool receives a new appcast from a vehicle/node it has never heard from before, it responds by posting an appcast request back to that vehicle for all applications, e.g., `APPCAST_REQ="node=henry,app=all, ..."`. This request will time-out shortly, but is usually sufficient to learn about all applications on that node. Subsequent requests are then more selective.

12 pHHostInfo: Detecting and Sharing Host Info

The `pHostInfo` application is a tool with a simple objective - determine the IP address of the machine on which it is running and post it to the MOOSDB. Although this information is available in a number of ways to a user at the keyboard, it may not be readily available for reasoning about within a MOOS community. Often, from an application's perspective, the host name is simply known and configured as `localhost`. This is fine for most purposes, but in situations where a user is on a machine where the IP address changes frequently, and the user is launching MOOS processes that talk to other machines, it may be very convenient to auto-determine the prevailing IP address and publish it to the MOOSDB. The typical usage scenario for `pHostInfo` is shown in Figure 69.

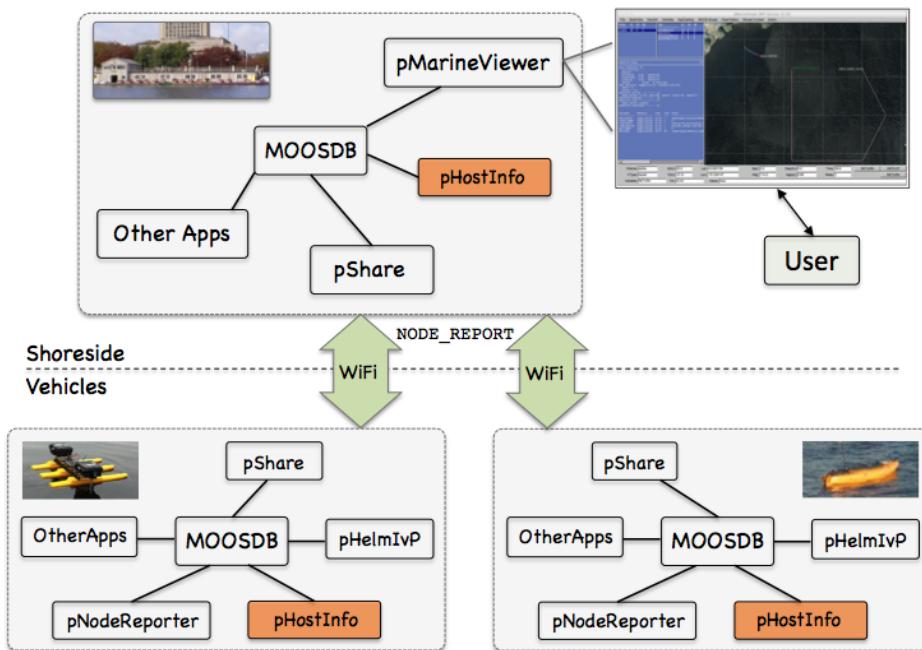


Figure 69: **Typical pHostInfo Topology:** A shoreside or topside community is receiving information from several deployed vehicles, in the form of node reports. The node reports contain time-stamped updated vehicle positions, from which the speed and distance measurements are derived and posted to the shoreside MOOSDB.

There are two scenarios where this is currently envisioned to be useful. The first is when the fielded vehicles are on the network with their IP addresses set via DHCP. For example if on the network via a cellular phone connection. The second is if the "vehicle" is a simulated vehicle running on a user's laptop also with its IP address set via DHCP. In both cases there may be another MOOS community with a known IP address, e.g., a shoreside community, to which the local vehicle wishes to inform it of its current IP address. This simple process however does not get involved in any activity regarding the communication to other MOOS communities, but simply tries to determine and post, the IP address, e.g., `PHI_HOST_IP = "192.168.0.1"` for other applications to do as they see fit.

12.1 Configuration Parameters for pHostInfo

The `pHostInfo` application may be configured with a configuration block within a MOOS mission file, typically with a `.moos` file suffix. The following parameters are defined for `pHostInfo`.

Listing 12.1: Configuration Parameters for pHostInfo.

- `temp_file_dir`: Directory where temporary files are written. Default is `"~/"`.
- `default_hostip`: IP address used if no IP address can otherwise be determined.
- `default_hostip_force`: This IP address will override any IP address from an auto-discovered network interface. Useful for debugging.

An Example MOOS Configuration Block

An example MOOS configuration block may be obtained from the command line with the following:

```
$ pHostInfo --example or -e
```

Listing 12.2: Example configuration of the pHostInfo application.

```
1 =====
2 pHostInfo Example MOOS Configuration
3 =====
4
5 ProcessConfig = pHostInfo
6 {
7     AppTick    = 4
8     CommsTick = 4
9
10    temp_file_dir  = ./
11    default_hostip = 192.168.0.55      // default is "localhost"
12
13    default_hostip_force = 192.168.0.99
14 }
```

12.2 Publications and Subscriptions for pHostInfo

The interface for `pHostInfo`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pHostInfo --interface or -i
```

12.2.1 Variables Published by pHostInfo

The primary output of `pHostInfo` to the MOOSDB are the following four variables. Once these variables are published, `pHostInfo` does not publish them again unless requested by receiving mail

`HOST_INFO_REQUEST`. Thus `pHostInfo` is mostly idle once the below four variables are posted.

- `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.
- `PHI_HOST_IP`: The single best guess of the Host's IP address.
- `PHI_HOST_IP_ALL`: A comma-separated list of IP addresses if multiple addresses detected.
- `PHI_HOST_IP_VERBOSE`: A comma-separated list of IP addresses, with source information, if multiple addresses detected.
- `PHI_HOST_PORT_DB`: The port number of the MOOSDB for this community.
- `PHI_HOST_PORT_INFO`: A comma-separated list of parameter-value pairs describing all relevant aspects of the host.

12.2.2 Variables Subscribed for by `pHostInfo`

The `pHostInfo` application subscribes to the following MOOS variable:

- `APPCAST_REQ`: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 11.10.4.
- `HOST_INFO_REQUEST`: A request to re-determine and re-post the platform's host information.
- `PSHARE_INPUT_SUMMARY`: The input routes used by `pShare` for listening for incoming messages from other MOOSDBs.

12.2.3 Command Line Usage of `pHostInfo`

The `pHostInfo` application is typically launched with pAntler, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing:

```
$ pHostInfo --help or -h
```

Listing 12.3: Command line usage for the `pHostInfo` tool.

```
1 Usage: pHostInfo file.moos [OPTIONS]
2
3 Options:
4   --alias=<ProcessName>
5       Launch pHostInfo with the given process
6       name rather than pHostInfo.
7   --example, -e
8       Display example MOOS configuration block
9   --help, -h
10      Display this help message.
11   --version,-v
12      Display the release version of pHostInfo.
13
14 Note: If argv[2] is not of one of the above formats
15       this will be interpreted as a run alias. This
16       is to support pAntler launching conventions.
```

12.3 Usage Scenarios the pHostInfo Utility

12.3.1 Handling Multiple IP Addresses

It is possible that a machine has more than one valid IP address at any given time, e.g., if its ethernet cable is plugged in, and it has a wireless connection. In this case, `pHostInfo` will make a guess that the ethernet connection takes precedent, and it will report this in the variable `PHI_HOST_IP`. The full set of IP addresses can be found in the other postings. For example it may not be uncommon to see something like the following three postings at one time:

```
PHI_HOST_IP      = 118.10.24.23
PHI_HOST_IP_ALL  = 118.10.24.23,169.224.126.40
PHI_HOST_IP_VERBOSE = OSX_ETHERNET2=118.10.24.23,OSX_AIRPORT=169.224.126.40
```

12.4 A Peek Under the Hood

The `pHostInfo` application currently only works for GNU/Linux and Apple OS X. It determines the IP information by making a system call within C++. A system call when generated will act as if the argument were typed on the command line. In this case the system call is generated and the output is redirected to a file. In a second step, `pHostInfo` then tries to read the IP address information from those files.

In GNU/Linux, the system call is based on the `ifconfig` command. In OS X, the system call is based on the `networksetup` command. Rather than determining in `pHostInfo` whether the user is running in a GNU/Linux or OS X environment, the system calls for both are invoked. Presumably a system call on a command not found in the user's shell path will not generate something that is confusable with a valid IP address.

12.4.1 Temporary Files

The temporary files are written to the user's home directory by default. This may be changed with the `temp_file_dir` configuration parameter, for example, `temp_file_dir=/tmp`. The set of temporary files are put into a folder named `.phostinfo/`. The set of temporary files may look like:

```
$ cd ~/.phostinfo
$ ls -a .ipinfo*
.ipinfo_linux_ethernet.txt    .ipinfo_osx_airport.txt    .ipinfo_osx_ethernet2.txt
.ipinfo_osx_wifi.txt          .ipinfo_linux_wifi.txt    .ipinfo_osx_ethernet1.txt
.ipinfo_osx_ethernet.txt
```

Some of these files may be empty, or some may contain error output if one of the system commands was not found, or was given an improper argument. The `pHostInfo` app will try to parse all of them to find a valid IP address. If more than one IP address is found, then this handled in the manner described previously in Section 12.3.1.

12.4.2 Possible Gotchas

The system calls invoked by `pHostInfo` need to be in the users shell path. A typical user default environment would have these in their shell path anyway, but it may be worth checking if things

aren't working properly. Below is a list of commands that are run under the hood, and their probable locations on your system.

```
For Linux:  
/sbin/ifconfig  
/bin/grep  
/usr/bin/cut  
/usr/bin/awk  
/usr/bin/print  
For OS X:  
/usr/sbin/networksetup
```

13 uPokeDB: Poking the MOOSDB from the Command Line

13.1 Overview

The **uPokeDB** application is a lightweight process that runs without any user interaction for writing to (poking) a running **MOOSDB** with one or more variable-value pairs. It is run from a console window with no GUI. For example, the alpha example mission is normally kicked off by hitting the **DEPLOY** button. The same could be accomplished from the terminal with:

```
$ uPokeDB alpha.moos DEPLOY=true, MOOS_MANUAL_OVERRIDE=false
```

After accepting variable-value pairs from the command line, **uPokeDB** connects to the **MOOSDB**, displays the variable values prior to poking, performs the poke, displays the variable values after poking, and then disconnects from the **MOOSDB** and terminates. It also accepts a .moos file as a command line argument to grab the IP and port information to find the **MOOSDB** for connecting. Other than that, it does not read a **uPokeDB** configuration block from the .moos file.

Other Methods for Poking a MOOSDB

There are few other MOOS applications capable of poking a MOOSDB. The **uMS** (MOOS Scope) is an application for both monitoring and poking a MOOSDB. It is substantially more feature rich than **uPokeDB**, and depends on the FLTK library. The **iRemote** application can poke the MOOSDB by using the **CustomKey** parameter, but is limited to the free unmapped keyboard keys, and is good when used with some planning ahead. The latest versions of **uMS** and **iRemote** are maintained on the Oxford MOOS website. The **uTermCommand** application (Section 16) is a tool primarily for poking the MOOSDB with a pre-defined list of variable-value pairs configured in its .moos file configuration block. The user initiates each poke by entering a keyword at a terminal window. Unlike **iRemote** it associates a variable-value pair with a key *word* rather than a keyboard key. The **uTimerScript** application (Section 6) is another tool for poking the MOOSDB with a pre-defined list of variable-value pairs configured in its .moos file configuration block. Unlike **uTermCommand**, **uTimerScript** will poke the MOOSDB without requiring further user action, but instead executes its pokes based on a timed script. The **uMOOSPoke** application, written by Matt Grund, is similar in intent to **uPokeDB** in that it accepts a command line variable-value pair. **uPokeDB** has a few additional features described below, namely multiple command-line pokes, accepting a .moos file on the command-line, and a MOOSDB summary prior and after the poke.

13.2 Command-line Arguments of uPokeDB

The command-line invocation of **uPokeDB** accepts two types of arguments - a .moos file, and one or more variable-value pairs. The former is optional, and if left unspecified, will infer that the machine and port number to find a running **MOOSDB** process is **localhost** and port 9000. The **uPokeDB** process does not otherwise look for a **uPokeDB** configuration block in this file. The variable-value pairs are delimited by the '=' character as in the following example:

```
$ uPokeDB alpha.moos FOO=bar TEMP=98.6 MOTTO="such is life" TEMP_STRING:=98.6
```

Since white-space characters on a command line delineate arguments, the use of double-quotes must be used if one wants to refer to a string value with white-space as in the third variable-value pair above. The value *type* in the variable-value pair is assumed to be a double if the value is numerical, and assumed to be a string type otherwise. If one really wants to poke with a string type that happens to be numerical, i.e., the string “98.6”, then the “:=” separator must be used as in the last argument in the example above. If `uPokeDB` is invoked with a variable type different than that already associated with a variable in the MOOSDB, the attempted poke simply has no effect.

13.3 MOOS Poke Macro Expansion

The `uPokeDB` utility supports macro expansion for timestamps. This may be used to generate a proxy posting from another application that uses timestamps as part of its posting. The macro for timestamps is `@MOOSTIME`. This will expand to the value returned by the MOOS function call `MOOSTime()`. This function call is implemented to return UTC time. The following is an example:

```
$ uPokeDB file.moos FOOBAR=color:red,temp=blue,timestamp=@MOOSTIME
```

The above poke would result in a posting similar to:

```
FOOBAR = color:red,temp=blue,timestamp=10376674605.24
```

As with other pokes, if the macro is part of a posting of type double, the timestamp is treated as a double. The posting

```
$ uPokeDB file.moos TIME_OF_START=@MOOSTIME
```

would result in the posting of type double for the variable `TIME_OF_START`, assuming it has not been posted previously as a different type.

13.4 Providing the ServerHost and ServerPort on the Command Line

The specification of a MOOS file on the command line is optional. The only two pieces of information `uPokeDB` needs from this file are (a) the `server_host` IP address, and (b) the `server_port` number of the running MOOSDB to poke. These values can instead be provided on the command line:

```
$ uPokeDB FOO=bar host=18.38.2.158 port=9000
```

If the `host` or the `port` are not provided on the command line, and a MOOS file is also not provided, the user will be prompted for the two values. Since the most common scenario by convention has the MOOSDB running on the local machine (“localhost”) with port 9000, these are the default values and the user can simply hit the return key.

```
$ uPokeDB FOO=bar // User launches with no server host/port info
$ Enter Server: [localhost] // User accepts default by hitting Return key
$   The server is set to "localhost" // Server host confirmed to be set to "localhost"
$ Enter Port: [9000] 9123 // User overrides the default 9000 port with 9123
$   The port is set to "9123" // Server port confirmed to be set to "9123"
```

13.5 Session Output from uPokeDB

The output in Listing 1 shows an example session when a running MOOSDB is poked with the following invocation:

```
$ uPokeDB alpha.moos DEPLOY=true RETURN=true
```

Lines 1-16 are standard output of a MOOS application that has successfully connected to a running MOOSDB. Lines 19-23 indicate the value of the variables prior to being poked, along with their source, i.e., the MOOS process responsible for publishing the current value to the MOOSDB, and the time at which it was last written. The time is given in seconds elapsed since the MOOSDB was started. Lines 26-30 show the new state of the poked variables in the MOOSDB after `uPokeDB` has done its thing.

Listing 13.1: An example uPokeDB session output.

```
1 -----
2 |      This is an Asynchronous MOOS Client      |
3 |      c. P. Newman U. Oxford 2001-2012      |
4 -----
5
6 -----MOOS CONNECT-----
7   contacting a MOOS server localhost:9000 -  try 00001
8   Contact Made
9   Handshaking as "uPokeDB"..... [OK]
10 -----
11
12 uPokeDB is Running:
13   +Baseline AppTick @ 5.0 Hz
14   +Comms is Full Duplex and Asynchronous
15   +Iterate Mode 0 :
16     -Regular iterate and message delivery at 5 Hz
17
18
19 PRIOR to Poking the MOOSDB
20   VarName          (S)ource    (T)ime      VarValue
21   -----          -----      -----      -----
22   DEPLOY           uTimerScript1.92    "true"
23   RETURN           pHelmIpvP       1         "false"
24
25
26 AFTER Poking the MOOSDB
27   VarName          (S)ource    (T)ime      VarValue
28   -----          -----      -----      -----
29   DEPLOY           uPokeDB        22.58     "true"
30   RETURN           uPokeDB        22.58     "false"
```

13.6 Publications and Subscriptions for uPokeDB

Variables published by the uPokeDB application

- **USER-DEFINED:** The only variables published are those that are poked. These variables are

provided on the command line. See Section [13.2](#).

Variables subscribed for by the uPokeDB application

- **USER-DEFINED:** Since uPokeDB provides two reports as described in the above Section [13.5](#), it subscribes for the same variables it is asked to poke, so it can generate its before-and-after reports.

14 pEchoVar: Re-publishing Variables Under a Different Name

14.1 Overview

The pEchoVar application is a lightweight process that runs without any user interaction for "echoing" the posting of specified variable-value pairs with a follow-on posting having different variable name. For example the posting of `F00=5.5` could be echoed such that `BAR=5.5` immediately follows the first posting. The motivation for this tool was convert, for example, a posting such as `GPS_X` to become `NAV_X`. The former is the output of a particular device, and the latter is a de facto standard for representing the vehicle's longitudinal position in local coordinates.

14.2 Using pEchoVar

Configuring `pEchoVar` minimally involves the specification of one or more `echo` or `flip` mapping events. It may also optionally involve specifying one or more logic conditions that must be met before mapping events are posted.

14.2.1 Configuring Echo Mapping Events

An *echo event mapping* maps one MOOS variable to another. Each mapping requires one line using the `echo` configuration parameter of the form:

```
echo = <MOOSVar> -> <MOOSVar>
```

The source `<MOOSVar>` and target `<MOOSVar>` components are case sensitive since they are MOOS variables. A source variable can be echoed to more than one target variable. If the set of lines forms a cycle, this will be detected and `pEchoVar` post a configuration and run warning and cease to perform any function whatsoever. An example configuration is given in Listing 1.

Listing 14.1: An example pEchoVar configuration block.

```
1 //-----
2 // pEchoVar configuration block
3
4 ProcessConfig = pEchoVar
5 {
6     AppTick    = 20
7     CommsTick = 20
8
9     echo = GPS_X          -> NAV_X
10    echo = GPS_Y         -> NAV_Y
11    echo = COMPASS_HEADING -> NAV_HEADING
12    echo = GPS_SPEED      -> NAV_SPEED
13 }
```

14.2.2 Configuring Flip Mapping Events

The `pEchoVar` application can be used to "flip" a variable rather than doing a simple echo. A flipped variable, like an echoed variable, is one that is republished under a different name, but a flipped

variable parses the contents of a string comprised of a series of `variable=value` comma-separated pairs, and republishes a portion of the series under the new variable name. For example, the following string,

```
ALPHA = "xpos=23, ypos=-49, depth=20, age=19.3, certainty=1"
```

may be flipped to publish the below new string, with the fields `xpos`, `ypos`, and `depth` replaced with `x`, `y`, `vehicle_depth` respectively.

```
BRAVO = "x=23, y=-49, vehicle_depth=20"
```

The above "flip relationship" is configured with the `flip` configuration parameter with the following form:

```
flip:<key> = source_variable = <variable>
flip:<key> = dest_variable = <variable>
flip:<key> = source_separator = <separator>
flip:<key> = dest_separator = <separator>
flip:<key> = filter = <variable>=<value>
flip:<key> = component = <old-field> -> <new-field>
flip:<key> = component = <old-field> -> <new-field>
```

The relationship is distinguished with a `<key>`, and several components. The `source_variable` and `dest_variable` components are mandatory and must be different. The `source_separator` and `dest_separator` components are optional with default values being the string `,`. Fields in the source variable will only be included in the destination variable if they are specified in a component mapping `<old-field> -> <new-field>`. The example configuration in Listing 2 implements the above described example flip mapping. In this case only postings that satisfy the further filter, `certainty=1`, will be posted.

Listing 14.2: An example pEchoVar configuration block with flip mappings.

```
1 //-----
2 // pEchoVar configuration block
3
4 ProcessConfig = pEchoVar
5 {
6     AppTick    = 10
7     CommsTick = 10
8
9     flip:1    = source_variable = ALPHA
10    flip:1   = dest_variable  = BRAVO
11    flip:1   = source_separator = ,
12    flip:1   = dest_separator = ,
13    flip:1   = filter        = certainty=1
14    flip:1   = component     = ypos -> y
15    flip:1   = component     = xpos -> x
16 }
```

Some caution should be noted with flip mappings - the current implementation does not check for cycles, as is done with echo mappings.

14.2.3 Applying Conditions to the Echo and Flip Operation

The execution of the mappings configured in `pEchoVar` may be configured to depend on one or more logic conditions. If conditions are specified in the configuration block, all specified logic conditions must be met or else the posting of echo and flip mappings will be suspended. The logic conditions are configured with the `condition` parameter 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. If a `condition` parameter is specified, `pEchoVar` will automatically subscribe to all MOOS variables used in the condition expressions.

14.2.4 Holding Outgoing Messages Until Conditions are Met

If the conditions are not met, all incoming mail messages that would otherwise result in an echo or flip posting, are held. When or if the conditions are met at some point later, those mail messages are processed in the order received and echo and flip mappings may be posted en masse. However, if several mail messages for a given MOOS variable are received and stored while conditions are unmet, only the latest received mail message for that variable will be processed. As an example, consider `pEchoVar` configured with the below two lines:

```
echo      = FOO -> BAR
condition = DEGREES <= 32
```

If the condition is not met for some period of time, and the following mail were received during this interval: `FOO="apples"`, `FOO="pears"`, `FOO="grapes"`, followed by `DEGREES=30`, then `pEchoVar` would post `BAR="grapes"` immediately on the very iteration that the `DEGREES=30` message was received. Note that `BAR="apples"` and `BAR="pears"` would never be posted. This is to help ensure that the `pEchoVar` memory doesn't grow unbounded by holding onto all mail while conditions are unmet.

The user may alternatively configure `pEchoVar` to *not* hold incoming mail messages when or if it is in a state where its logic conditions are not met. This can be done with the `hold_messages` parameter:

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

When configured this way, upon meeting the specified logic conditions, `pEchoVar` will begin processing echo and flip mappings when or if new mail messages are received relevant to the mappings. In the above example, once `DEGREES=30` is received by `pEchoVar`, nothing would be posted until new incoming mail on the variable `FOO` is received (not even `BAR="grapes"`).

14.2.5 Limiting the Echo Posting Frequency to the AppTick Setting

By default, when the conditions are met, an echo posting is made once for each incoming piece of mail related that echo mapping. If **FOO** is echo mapped to **BAR**, and if 40 pieces of incoming mail for **FOO** are received on one iteration, 40 postings are made to **BAR** on that iteration. Instead one may wish that, on each iteration where there is posting ready for **BAR**, that only the latest value for **BAR** be made. This may be arranged with the `echo_latest_only` parameter:

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

In this case, the frequency of postings to **BAR** and all other echo mappings will occur at most at a frequency equal to the `AppTick` setting.

14.3 Configuring for Vehicle Simulation with pEchoVar

When in simulation mode with uSimMarine, the navigation information is generated by the simulator and not the sensors such as GPS or compass as indicated in lines 9-12 in Listing 1. The simulator instead produces **USM_*** values which can be echoed as **NAV_*** values as shown in Listing 3.

Listing 14.3: An example pEchoVar configuration block during simulation.

```
1 //-----
2 // pEchoVar configuration block (for simulation mode)
3
4 ProcessConfig = pEchoVar
5 {
6   AppTick    = 20
7   CommsTick = 20
8
9   echo = USM_X      -> NAV_X
10  echo = USM_Y      -> NAV_Y
11  echo = USM_HEADING -> NAV_HEADING
12  echo = USM_SPEED   -> NAV_SPEED
13 }
```

Note in more recent versions of **uSimMarine** the simulator output may be changed to have a **NAV_** prefix by setting `prefix = NAV_`, obviating the use of **pEchoVar** configured as above.

14.4 Configuration Parameters for pEchoVar

The following parameters are defined for **pEchoVar**. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so.

Listing 14.4: Configuration Parameters for pEchoVar.

echo: A mapping from one MOOS variable to another constituting an echo.
Section 14.2.1.

- echo_latest_only:** If true, only the latest value of variable will be echoed on each iteration, even if several pieces of incoming mail have been received since the last posting. Legal values: true, false. The default is false. Section [14.2.1](#).
- condition:** A logic condition that must be met or all echo and flip publications are held. Section [14.2.3](#).
- flip:** A description of how components from one variable are re-posted under another MOOS variable. Section [14.2.2](#).
- hold_messages:** If true, messages are held when conditions are not met for later processing when logic conditions are indeed met. Legal values: true, false. The default is true. Section [14.2.3](#).

14.5 Publications and Subscriptions for pEchoVar

The interface for `pEchoVar`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pEchoVar --interface or -i
```

14.5.1 Variables Posted by pEchoVar

- **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](#).
- **<USER-DEFINED>:** Any MOOS variable specified in either the `echo` or `flip` config parameters.

14.5.2 Variables Subscribed for by pEchoVar

- **APPCAST_REQ:** A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section [11.10.4](#).
- **<USER-DEFINED>:** Any MOOS variables found in the antecedent of either the `echo` or `flip` mappings. It will also subscribe for any MOOS variable found in any of its logic conditions.

14.6 Terminal and AppCast Output

The `pEchoVar` application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 5 below. This application is also appcast enabled, meaning its reports are published to the MOOSDB and viewable from any uMAC application or pMarineViewer. See Section [10](#) for more on appcasting and viewing appcasts. The counter on the end of line 2 in parentheses is incremented on each iteration of `pEchoVar`, and serves a bit as a heartbeat indicator. The "0/0" also on line 2 indicates there are no configuration or run warnings detected.

Listing 14.5: Example terminal or appcast output for pEchoVar.

¹ =====

```

2 pEchoVar alpha                                0/0(81)
3 =====
4 conditions_met:  true
5 hold_messages:  true
6 echo_latest_only: false
7
8 =====
9 Echoes: (5)
10 =====
11
12 Source          Dest        Hits  Posts
13 -----  -----
14 NAV_X    --> NAV_XX      324   324
15 NAV_X    --> NAV_XPOS    324   324
16 NAV_Y    --> NAV_YY      324   324
17 NAV_Y    --> NAV_YPOS    324   324
18 NAV_SPEED --> NAV_SPEED_ALT 324   324
19
20 =====
21 Flips: (1)
22 =====
23
24                               Src  Dest          Old  New
25 Key  Hits  Source          Dest  Sep  Sep  Filter  Field  Field
26 ---  ----  -----          -----  ---  ---  -----  -----  -----
27 1    162  NODE_REPORT_LOCAL  FOOBAR ,  #  type:kayak  X  xpos
28 1    162  NODE_REPORT_LOCAL  FOOBAR ,  #  type:kayak  Y  ypos

```

Lines 4 indicates whether or not any specified logic conditions have been met. This line will also read `true` even if no logic conditions were provided. Lines 5 and 6 simply confirm the user's settings for the `hold_messages` and `echo_latest_only` parameters discussed in Sections 14.2.4 and 14.2.5 respectively.

Lines 12-18 convey the configured echo mappings, one for each line. At the end of each line, the *Hits* column shows the number of incoming mails received for that variable. The number of times it is echoed, or re-posted is shown under the *Posts* column. When `echo_latest_only` is false, these numbers should match. Lines 20-28 convey the configure flips. A single flip configuration, identified by its key, may have several lines, as in this example. Here the only difference between lines 27 and 28 are the flip components. One maps `X` to `xpos`, and the other maps `Y` to `ypos`.

The terminal or appcast output shown in Listing 5 above may be seen first hand by running the Alpha example mission. The below configuration block in Listing 6 corresponds to the above appcast output. The user just needs to add `pEchoVar` to the Antler launch list.

Listing 14.6: Example pEchoVar configuration from the Alpha example mission.

```

1 ProcessConfig = pEchoVar
2 {
3     AppTick = 1
4     CommsTick = 1
5
6     echo = NAV_X      -> NAV_XX

```

```

7 echo = NAV_X      -> NAV_XPOS
8 echo = NAV_Y      -> NAV_YY
9 echo = NAV_Y      -> NAV_YPOS
10 //echo = NAV_YY   -> FOOBAR
11 //echo = FOOBAR   -> NAV_Y
12 echo = NAV_SPEED -> NAV_SPEED_ALT
13
14 FLIP:1    = source_variable  = NODE_REPORT_LOCAL
15 FLIP:1    = dest_variable   = FOOBAR
16 FLIP:1    = source_separator = ,
17 FLIP:1    = dest_separator  = #
18 FLIP:1    = filter = type == kayak
19 FLIP:1    = component = X -> xpos
20 FLIP:1    = component = Y -> ypos
21 }

```

The two echo mappings in lines 10 and 11 may be commented out to demonstrate the detection of echo mapping cycles. The pairs of mappings in Lines 6-7 and 8-9 demonstrate that a single incoming variable may be mapped to multiple destinations.

15 pSearchGrid: Using a 2D Grid Model for Track History

The `pSearchGrid` application is a module for storing a history of vehicle positions in a 2D grid defined over a region of operation. This module may have utility as-is, to help guide a vehicle to complete or uniform coverage of a given area, but also exists as an example of how to use and visualize the `XYConvexGrid` data structure. This data structure may be used in similar modules to store a wide variety of user specified data for later use by other modules or simply for visualization. Here the structure is used in a MOOS application, but it could also be used within a behavior. The `pSearchGrid` module begins with a user-defined grid, defined in the MOOS configuration file. As a vehicle moves and generates node reports, `pSearchGrid` simply notes the vehicle's current position increments the cell containing vehicle's current position. After time, the grid shows a cumulative history of the most commonly traveled positions in the local operating area.

The `pSearchGrid` module is an optional part of the Charlie example mission discussed later in this section. When running and grid viewing is enabled in `pMarineViewer`, the viewer may show something similar to Figure 70.

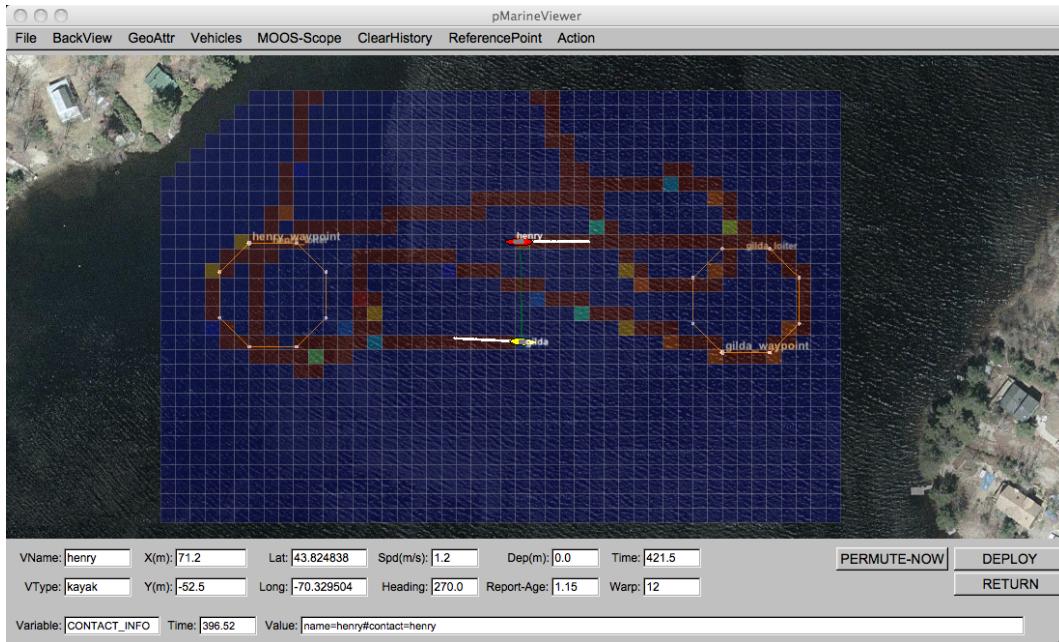


Figure 70: **An Example pSearchGrid Scenario:** A grid is configured around the operation area of the Berta example mission. Higher values in a given grid cell represents a longer noted time of any vehicle passing through that cell.

15.1 Using pSearchGrid

The present configuration of pSearchGrid will store values in its grid cells proportional to the time a vehicle is noted to be within a grid cell. One may use this application as written or regard this as a template for writing a new application that stores some other value with each grid cell, such as bathymetry data, water velocity data, or likelihood of there being an object of interest in the region

of the cell for further investigation. The below discusses the general usage of managing the grid data within a MOOS application.

15.1.1 Basic Configuration of Grid Cells

Basic grid configuration consists of specifying (a) a convex polygon, and (b) the size of the grid squares. From this the construction of a grid proceeds by calculating a bounding rectangle containing the polygon, generating a set of squares covering the rectangle, and then removing the squares not intersecting the original polygon. The result is a non-rectilinear grid as shown in Figure 70. The basic configuration of grid cells is done with the `grid_config` parameter specifying the points and the cell size as the following example:

```
grid_config = pts={-50,-40:-10,0:180,0:180,-150:-50,-150}, cell_size=5
```

15.1.2 Cell Variables

The grid is primarily used for associating information with each grid cell. In `pSearchGrid`, the grid is configured to store a single numerical value (a C++ double) with each cell. The grid structure may be configured in another application to store multiple numerical values with each cell. In `pSearchGrid`, the cell variables are declared upon startup in the MOOS configuration block, similar to:

```
grid_config = cell_vars=x:0y:100
grid_config = cell_min=x:0
grid_config = cell_min=x:100
```

This configuration associates two cell variables, x and y , which each cell. The cell variable x is initialized to 0 for each cell, and the cell variable y is initialized to 100. The first variable has a minimum and maximum constraint of 0 and 100, and the second variable is unconstrained. The above configuration could also have been made on one line, separating each with a comma.

15.1.3 Serializing and De-serializing the Grid Structure

The grid structure maintained by `pSearchGrid` is periodically published to the MOOSDB for consumption by other applications, or conceivably by a behavior with the IvP Helm. The structure may be serialized into a string by calling the `get_spec()` method on an instance of the `XYConvexGrid` class. Serializing a grid instance and posting it to the MOOSDB may look something like:

```
#include "XYConvexGrid.h"
....
XYConvexGrid mygrid;
....
string str = mygrid.get_spec();
m_Comms.Notify(VIEW_GRID, str);
```

Likewise a string representation of the grid may be de-serialized to a grid instance by calling a function provided in the same library where the grid is defined. De-serializing a string read from the MOOSDB into a local grid instance may look something like:

```

#include "XYFormatUtilsConvexGrid.h"
#include "XYConvexGrid.h"
...
string grid_string_spec;
...
XYConvexGrid mygrid = string2ConvexGrid(grid_string_spec);

```

The same function used to de-serialize a string is used by `pSearchGrid` to configure the initial grid. In other words, the components from the various GRID configuration parameters are concatenated into a single comma-separated string and passed to the de-serialization function, `string2ConvexGrid`, to form the initial instance used by `pSearchGrid`.

15.1.4 Resetting the Grid

The grid may be reset at any point in the operation when `pSearchGrid` receives mail on the variable `PSG_GRID_RESET`. If this variable's string value is the empty string, it will reset all cell variables for all cell elements to their given initial values. If the string value is non-empty, it will interpret this as an attempt to reset the values for a named cell variable. Thus `PSG_GRID_RESET="x"` will reset the `x` cell variable and no other cell variables. If "`x`" is not a cell variable, no action will be taken.

15.1.5 Viewing Grids in pMarineViewer

The `pMarineViewer` will display grids by subscribing for postings to the variable `VIEW_GRID`. As with other viewable objects such as polygons, points, etc., the viewer keeps a local cache of grid instances, one for each named grid, based on the grid label. For example if two successive grids are received with different labels, the viewer will store and render both of them. If they have the same label, the second will replace the first and the viewer will just render the one.

The rendering of grids may be toggled on/off by hitting the 'g' key, and may be made more transparent with the CTRL-g key, and less transparent with the ALT-g key. The color map used for the grid is taken from the typical MATLAB color map; red is the highest value, blue is the lowest.

15.1.6 Examples

Example usage of `pSearchGrid` may be found in both the Charlie and Berta example missions distributed with the moos-ipv tree. For example, in the Charlie example mission, edit `charlie.moos` and uncomment the line beginning with `Run = pSearchGrid`. Likewise for the Berta mission and the file `meta_shoreside.moos`.

15.2 Configuration Parameters of pSearchGrid

The following parameters are defined for `pSearchGrid`. A more detailed description is provided in other parts of this section. Parameters having default values indicate so.

Listing 15.1: Configuration Parameters for pSearchGrid.

`grid_config:` A portion or all of a grid configuration description. See Listing 2.

An Example MOOS Configuration Block

An example `pSearchGrid` configuration block is given in Listing 2 below, and may also be seen from the command line invocation of:

```
$ pSearchGrid --example or -e
```

Listing 15.2: Example configuration of the `pSearchGrid` application.

```
1 =====
2 pSearchGrid Example MOOS Configuration
3 =====
4
5 ProcessConfig = pSearchGrid
6 {
7     AppTick    = 4
8     CommsTick = 4
9
10    GRID_CONFIG = pts={-50,-40: -10,0: 180,0: 180,-150: -50,-150}
11    GRID_CONFIG = cell_size=5
12    GRID_CONFIG = cell_vars=x:0:y:0:z:0
13    GRID_CONFIG = cell_min=x:0
14    GRID_CONFIG = cell_max=x:10
15    GRID_CONFIG = cell_min=y:0
16    GRID_CONFIG = cell_max=y:1000
17 }
```

15.3 Publications and Subscriptions for `pSearchGrid`

The interface for `pSearchGrid`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pSearchGrid --interface or -i
```

15.3.1 Variables Published by `pSearchGrid`

The `pSearchGrid` application publishes the following variables:

- **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 11.10.4.
- **VIEW_GRID**: A full description of a grid format and contents.

A typical string may be:

```
VIEW_GRID = pts={-50,-40:-10,0:100,0:100,-100:50,-150:-50,-100},cell_size=5,
           cell_vars=x:0:y:0:z:0,cell_min=x:0,cell_max=x:50,cell=211:x:50,
           cell=212:x:50,cell=237:x:50,cell=238:x:50,label=psg
```

15.3.2 Variables Subscribed for by pSearchGrid

The `pSearchGrid` application subscribes to the following MOOS variables:

- `APPCAST_REQ`: A request to generate and post a new appcast report, with reporting criteria, and expiration. Section 11.10.4.
- `NODE_REPORT`: A node report for a given vehicle from `pNodeReporter`.
- `NODE_REPORT_LOCAL`: A node report for a given vehicle from `pNodeReporter`.
- `PSG_GRID_RESET`: A request to reset the grid to its original configuration.

15.3.3 Command Line Usage of pSearchGrid

The `pSearchGrid` application is typically launched with pAntler, along with a group of other modules. However, it may be launched separately from the command line. The command line options may be shown by typing "`pSearchGrid --help`":

Listing 15.3: Command line usage for the `pSearchGrid` tool.

```
1 Usage: pSearchGrid file.moos [OPTIONS]
2
3 Options:
4   --alias=<ProcessName>
5     Launch pSearchGrid with the given process
6     name rather than pSearchGrid.
7   --example, -e
8     Display example MOOS configuration block
9   --help, -h
10    Display this help message.
11   --version,-v
12    Display the release version of pSearchGrid.
13 Note: If argv[2] is not of one of the above formats
14      this will be interpreted as a run alias. This
15      is to support pAntler launching conventions.
```

16 uTermCommand: Poking the MOOSDB with Pre-Set Values

The `uTermCommand` application is a terminal based tool for poking the MOOS database with pre-defined variable-value pairs. This can be used for command and control for example by setting variables in the MOOSDB that affect the behavior conditions running in the helm. There are a few other ways of doing this:

- *pMarineViewer*: The action pull-down menu and on-screen buttons may be used for posting the MOOSDB. See Section 2.7.
- *uPokeDB*: A command-line tool for poking the MOOSDB. This may be used in conjunction with shell aliases or shell scripts for further convenience. See Section 13.
- *iRemote*: The custom keys feature may be used to bind variable-value pairs to the numeric keys. The primary drawback is the limitation to ten mappings.

16.1 Configuration Parameters for uTermCommand

The variable-value mappings are set in the `uTermCommand` configuration block of the MOOS file. Each mapping requires one line of the form:

```
cmd = cue --> variable --> value
```

The *cue* and *variable* fields are case sensitive, and the *value* field may also be case sensitive depending on how the subscribing MOOS process(es) handle the value. An example configuration is given in Listing 1.

Listing 16.1: An example uTermCommand configuration block.

```
1 //-----
2 // uTermCommand configuration block
4
5 ProcessConfig = uTermCommand
6 {
7     cmd = override_true    --> MOOS_MANUAL_OVERRIDE --> true
8     cmd = override_false   --> MOOS_MANUAL_OVERRIDE --> false
9     cmd = deploy_true      --> DEPLOY          --> true
10    cmd = deploy_false     --> DEPLOY          --> false
11    cmd = return_true      --> RETURN          --> true
12    cmd = return_false     --> RETURN          --> false
13 }
```

Recall the type of a MOOS variable is either a string, double or binary data. If a variable has yet to be posted to the MOOSDB, it accepts whatever type is first written, otherwise postings of the wrong type are ignored. If quotes surround the entry in the value field, it is interpreted to be a string. If not, the value is inspected as to whether it represents a numerical value. If so, it is posted as a double. For example `true` and "true" are the same type (no such thing as a Boolean type), 25 is a double and "25" is a string.

16.2 Run Time Console Interaction

When `uTermCommand` is launched, a separate thread accepts user input at the console window. When first launched the entire list of cues and the associated variable-value pairs are listed. Listing 2 shows what the console output would look like given the configuration parameters of Listing ???. This configuration block is in the Alpha example mission. You can launch that mission and then launch `uTermCommand`:

```
$ cd moos-ivp/ivp/missions/s1_alpha  
$ ./launch.sh
```

Then in a separate terminal:

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

The output in the terminal window should look similar to Listing 2. Note that even though quotes were not necessary in the configuration file to clarify that `true` was to be posted as a string, the quotes are always placed around string values in the terminal output.

Listing 16.2: Console output at start-up.

1	Cue	VarName	VarValue
2	-----	-----	-----
3	override_true	MOOS_MANUAL_OVERRIDE	"true"
4	override_false	MOOS_MANUAL_OVERRIDE	"false"
5	deploy_true	DEPLOY	"true"
6	deploy_false	DEPLOY	"false"
7	return_true	RETURN	"true"
8	return_false	RETURN	"false"
9			
10	>		

In the Alpha mission, the DEPLOY button in the lower right corner of `pMarineViewer` is configured to post:

```
MOOS_MANUAL_OVERRIDE = false  
DEPLOY = true
```

This will be handled instead by `uTermCommand` in our simple example. A prompt is shown on the last line where user key strokes will be displayed. As the user types characters, the list of choices is narrowed based on matches to the cue. After typing a single 'o' character, only the `override_true` and `override_false` cues match and the list of choices shown are reduced as shown in Listing 3. At this point, hitting the TAB key will complete the input field out to `override_`, much like tab-completion works at a Linux shell prompt.

Listing 16.3: Console output after typing a single character 'r'.

```

1   Cue
2 -----
3   override_true      VarName
4   override_false     VarName
5
6   > o

```

When the user has typed out a valid cue that matches a single entry, only the one line is displayed, with the tag `<-- SELECT` at the end of the line, as shown in Listing 4.

Listing 16.4: Console output when a single command is identified.

```

1   Cue
2 -----
3   override_false      VarName
4
5   > override_false

```

At this point hitting the ENTER key will execute the posting of that variable-value pair to the MOOSDB, and the console output will return to its original start-up output. A local history is augmented after each entry is made, and the up- and down-arrow keys can be used to select and re-execute postings on subsequent iterations. To finish the launch in the Alpha mission, use `uTermCommand` to post `DEPLOY=true`.

16.3 Connecting uTermCommand to the MOOSDB Under an Alias

A convention of MOOS is that each application connected to the MOOSDB must register with a unique name. Typically the name used by a process to register with the MOOSDB is the process name, e.g., `uTermCommand`. One may want to run multiple instances of `uTermCommand` all connected to the same MOOSDB. To support this, an optional command line argument may be provided when launching `uTermCommand`:

```
$ uTermCommand file.moos --alias=uTermCommandAlpha
```

The command line argument may also be invoked from within `pAntler` to launch multiple `uTermCommand` instances simultaneously. The configuration block in the mission file needs to have the same name as the launch alias.

16.4 Publications and Subscriptions for uTermCommand

The only variables published by `uTermCommand` are those configured and selected by the user at run-time, and `uTermCommand` does not subscribe for any variables.

17 uSimCurrent: Simulating Drift Effects

The `uSimCurrent` MOOS application is a newcomer in the toolbox and documentation is thin. Nevertheless it has been tested and used quite a bit and is worth a quick introduction here for those with a need for some ability to simulate water current on unmanned vehicles.

`uSimCurrent` is intended to be used with the `uSimMarine` simulator, by generating drift vectors and publishing them to the MOOSDB. The `uSimMarine` simulator has a generic interface to accept externally published drift vectors regardless of the source, written to the variables `DRIFT_X`, `DRIFT_Y`, and `DRIFT_VECTOR`, as described in Section 9.6. The `uSimCurrent` application reads a provided *current field file* containing an association of water current to positions in the water. On iteration of `uSimCurrent`, the vehicle's current position is noted, looked up in the current-field data structure, and a new drift vector is posted. The idea is shown in Figure 71.

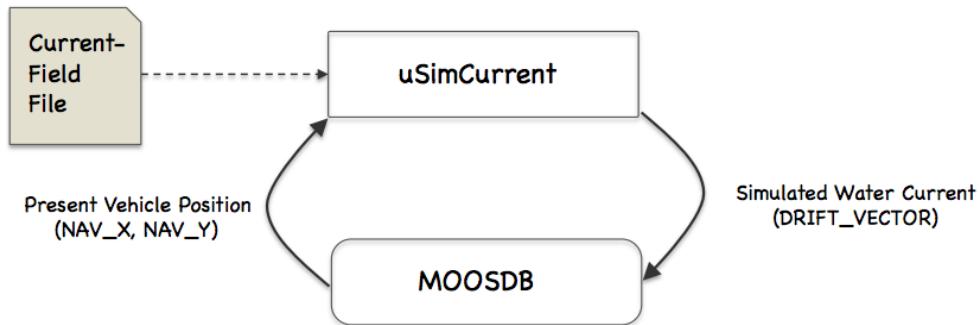


Figure 71: The `uSimCurrent` utility: The simulator is initialized with a data file describing currents and locations. The simulator then repeatedly publishes a current vector based on the present vehicle position.

17.1 Configuration Parameters for uSimCurrent

The following configuration parameters are defined for `uSimCurrent`. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so.

Listing 17.1: Configuration Parameters for `uSimCurrent`.

```
current_field: Name of a file describing a current field.  
current_field_active: Boolean indicating whether the simulator is active.
```

17.2 Publications and Subscriptions for uSimCurrent

The interface for `uSimCurrent`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uSimCurrent --interface or -i
```

17.2.1 MOOS Variables Published by uSimCurrent

The primary output of uSimCurrent to the MOOSDB is the drift vector to be consumed by the [uSimMarine](#) application.

- [DRIFT_VECTOR](#): drift vector representing the prevailing current. Section [9.6](#).
- [USC_CFIELD_SUMMARY](#): Summary of configured current field.
- [VIEW_VECTOR](#): Vector objects suitable for rendering in GUI applications.

17.2.2 MOOS Variables Subscribed for by uSimCurrent

Variables subscribed for by [uSimCurrent](#) are summarized below.

- [NAV_X](#): The ownship vehicle position on the x axis of local coordinates.
- [NAV_Y](#): The ownship vehicle position on the y axis of local coordinates.

18 The Alog-Toolbox for Analyzing and Editing Mission Log Files

18.1 Overview

The Alog-Toolbox is a set of five post-mission analysis utility applications `alogview`, `alogscan`, `alogrm`, `aloggrep`, `alogclip`. Each application manipulates or rendersings .alog files generated by the `pLogger` application. Three of the applications, `alogclip`, `aloggrep`, and `alogrm` are command-line tools for filtering a given .alog file to a reduced size. Reduction of a log file size may facilitate the time to load a file in a post-processing application, may facilitate its transmission over slow transmission links when analyzing data between remote users, or may simply ease in the storing and back-up procedures. The `alogscan` tool provides statistics on a given .alog file that may indicate how to best reduce file size by eliminating variable entries not used in post-processing. It also generates other information that may be handy in debugging a mission. The `alogview` tool is a GUI-based tool that accepts one or more .alog files and renders a vehicle positions over time on an operation area, provides time-correlated plots of any logged numerical MOOS variables, and renders helm autonomy mode data with plots of generated objective functions.

18.2 An Example .alog File

The .alog file used in the examples below was generated from the Alpha example mission. This file, `alpha.alog`, is found in the missions distributed with the MOOS-IvP tree. The `alpha.alog` file was created by simply running the mission as described, and can be found in:

```
moos-ivp/trunk/ivp/missions/alpha/alpha.alog.
```

18.3 The `alogscan` Tool

The `alogscan` tool is a command-line application for providing statistics relating to a given .alog file. It reports, for each unique MOOS variable in the log file, (a) the number of lines in which the variable appears, i.e., the number of times the variable was posted by a MOOS application, (b) the total number of characters comprising the variable value for all entries of a variable, (c) the timestamp of the first recorded posting of the variable, (d) the timestamp of the last recorded posting of the variable, (e) the list of MOOS applications the posted the variable.

18.3.1 Command Line Usage for the `alogscan` Tool

The `alogscan` tool is run from the command line with a given .alog file and a number of options. The usage options are listed when the tool is launched with the `-h` switch:

```
$ alogscan --help or -h
```

Listing 18.1: Command line usage for the `alogscan` tool.

```
1 Usage:  
2   alogscan file.alog [OPTIONS]  
3  
4 Synopsis:  
5   Generate a report on the contents of a given
```

```

6   MOOS .alog file.
7
8 Options:
9 --sort=type  Sort by one of SIX criteria:
10          start: sort by first post of a var
11          stop: sort by last post of a var
12  (Default) vars: sort by variable name
13          proc: sort by process/source name
14          chars: sort by total chars for a var
15          lines: sort by total lines for a var
16
17 --appstat    Output application statistics
18 -r,--reverse Reverse the sorting output
19 -n,--nocolors Turn off process/source color coding
20 -h,--help     Displays this help message
21 -v,--version  Displays the current release version
22
23 See also: aloggrp, alogrm, alogclip, alogview

```

The order of the arguments passed to `alogscan` do not matter. The lines of output are sorted by grouping variables posted by the same MOOS process or source, as in Listing 2 below. The sorting criteria can instead be done by alphabetical order on the variable name (`--sort=vars`), the total characters in the file due to a variable (`--sort=chars`), the total lines in the file due to a variable (`--sort=lines`), the time of the first posting of the variable (`--sort=start`), or the time of the last posting of the variable (`--sort=stop`). The order of the output may be reversed (`-r, --reverse`). By default, the entries are color-coded by the variable source, using the few available terminal colors (there are not many). When unique colors are exhausted, the color reverts back to the default terminal color in effect at the time.

18.3.2 Example Output from the `alogscan` Tool

The output shown in Listing 2 was generated from the `alpha.alog` file generated by the Alpha example mission.

Listing 18.2: Example output from the `alogscan` tool.

1 Variable Name	2 -----	3 Lines	4 Chars	5 Start	6 Stop	7 Sources
3 DB_CLIENTS		282	22252	-0.38	566.42	MOOSDB_alpha
4 DB_TIME		556	7132	1.21	566.18	MOOSDB_alpha
5 DB_UPTIME		556	7173	1.21	566.18	MOOSDB_alpha
6 USIMMARINE_STATUS		276	92705	0.39	565.82	uSimMarine
7 NAV_DEPTH		6011	6011	1.43	566.38	uSimMarine
8 NAV_HEADING		6011	75312	1.43	566.38	uSimMarine
9 NAV_LAT		6011	74799	1.43	566.38	uSimMarine
10 NAV_LONG		6011	80377	1.43	566.38	uSimMarine
11 NAV_SPEED		6011	8352	1.43	566.38	uSimMarine
12 NAV_STATE		6011	18033	1.43	566.38	uSimMarine
13 NAV_X		6011	72244	1.43	566.38	uSimMarine
14 NAV_Y		6011	77568	1.43	566.38	uSimMarine
15 NAV_YAW		6011	80273	1.43	566.38	uSimMarine
16 BHV_IPF		2009	564165	46.26	542.85	pHelmIvP
17 CREATE_CPU		2108	2348	46.26	566.33	pHelmIvP
18 CYCLE_INDEX		5	5	44.98	543.09	pHelmIvP
19 DEPLOY		3	14	3.84	543.09	pHelmIvP,pMarineViewer
20 DESIRED_HEADING		2017	5445	3.85	543.09	pHelmIvP
21 DESIRED_SPEED		2017	2017	3.85	543.09	pHelmIvP

```

22 HELM_IPF_COUNT          2108    2108   46.26  566.32 pHelmIvP
23 HSLINE                  1        3     3.84   3.84 pHelmIvP
24 IVPHELM_DOMAIN          1        29     3.84   3.84 pHelmIvP
25 IVPHELM_ENGAGED         462    3342   3.85  566.32 pHelmIvP
26 IVPHELM_MODESET          1        0     3.84   3.84 pHelmIvP
27 IVPHELM_POSTINGS         2014   236320  46.26  543.33 pHelmIvP
28 IVPHELM_STATEVARS        1        20    44.98  44.98 pHelmIvP
29 IVPHELM_SUMMARY          2113   612685  44.98  566.33 pHelmIvP
30 LOOP_CPU                 2108   2348   46.26  566.33 pHelmIvP
31 PC_hslide                1        9     44.98  44.98 pHelmIvP
32 PC_waypt_return           3        14    44.98  543.33 pHelmIvP
33 PC_waypt_survey            3        14    44.98  543.33 pHelmIvP
34 PHELMIVP_STATUS           255   198957   3.85  565.12 pHelmIvP
35 PLOGGER_CMD                1        17    3.84   3.84 pHelmIvP
36 PWT_BHV_HSLINE             1        1    44.98  44.98 pHelmIvP
37 PWT_BHV_WAYPT_RETURN        3        5    44.98  543.09 pHelmIvP
38 PWT_BHV_WAYPT_SURVEY        2        4    44.98  462.90 pHelmIvP
39 RETURN                     4        19    3.84  543.09 pMarineViewer
40 STATE_BHV_HSLINE             1        1    44.98  44.98 pHelmIvP
41 STATE_BHV_WAYPT_RETURN        4        4    44.98  543.33 pHelmIvP
42 STATE_BHV_WAYPT_SURVEY        3        3    44.98  463.15 pHelmIvP
43 SURVEY_INDEX                 10       10    44.98  429.70 pHelmIvP
44 SURVEY_STATUS                1116   77929   45.97  462.90 pHelmIvP
45 VIEW_POINT                   4034   101662   44.98  543.33 pHelmIvP
46 VIEW_SEGLIST                  4       273    44.98  543.33 pHelmIvP
47 WPT_INDEX                     1        1    463.15  463.15 pHelmIvP
48 WPT_STAT                      223   15626   463.15  543.09 pHelmIvP
49 LOGGER_DIRECTORY               56     1792    1.07  559.19 pLogger
50 PLOGGER_STATUS                 263   331114    1.07  566.40 pLogger
51 DESIRED_RUDDER                10185  150449   -9.28  545.18 pMarinePID
52 DESIRED_THRUST                 10637  20774   -9.28  566.52 pMarinePID
53 MOOS_DEBUG                     5       39    -9.31  545.23 pMarinePID,pHelmIvP
54 PMARINEPID_STATUS              279   81990    0.95  566.28 pMarinePID
55 HELM_MAP_CLEAR                  1        1    -1.56  -1.56 pMarineViewer
56 MOOS_MANUAL_OVERRIDE             1        5     44.65  44.65 pMarineViewer
57 PMARINEVIEWER_STATUS              270   95560   -0.95  564.89 pMarineViewer
58 NODE_REPORT_LOCAL                 1159   207535   1.15  565.91 pNodeReporter
59 PNODEREPORTER_STATUS              233   50534   -0.37  563.93 pNodeReporter
60 -----
61 Total variables: 57
62 Start/Stop Time: -9.31 / 566.52

```

When the `-appstat` command line option is included, a second report is generated, after the above report, that provides statistics keyed by application, rather than by variable. For each application that has posted a variable recorded in the given `.alog` file, the number of lines and characters are recorded, as well as the percentage of total lines and characters. An example for the `alpha.alog` file is shown in Listing 3.

Listing 18.3: Example `alogscan` output generated with the `-appstat` command line option.

63 MOOS Application	Total Lines	Total Chars	Lines/Total	Chars/Total
64 -----	-----	-----	-----	-----
65 MOOSDB_alpha	1394	36557	1.37	1.08
66 uSimMarine	54375	585674	53.57	17.29
67 pHelmIvP	22642	1825437	22.31	53.89
68 pLogger	319	332906	0.31	9.83
69 pMarinePID	21106	253252	20.80	7.48
70 pMarineViewer	279	95599	0.27	2.82
71 pNodeReporter	1392	258069	1.37	7.62

Further Tips

- If a small number of variables are responsible for a relatively large portion of the file size, and are expendable in terms of how data is being analyzed, the variables may be removed to ease the handling, transmission, or storage of the data. To remove variables from existing files, the `alogrm` tool described in 18.6 may be used. To remove the variable from future files, the `pLogger` configuration may be edited by either removing the variable from the list of variables explicitly requested for logging, or if WildCardLogging is used, mask out the variable with the `WildCardOmitPattern` parameter setting. See the `pLogger` documentation.
- The output of `alogscan` can be further distilled using common tools such as `grep`. For example, if one only wants a report on variables published by the `pHelmIpvP` application, one could type:

```
$ alogscan alpha.alog | grep pHelmIpvP
```

18.4 The `alogclip` Tool

The `alogclip` tool will prune a given `.alog` file based on a given beginning and end timestamp. This is particularly useful when a log file contains a sizeable stretch of data logged after mission completion, such as data being recorded while the vehicle is being recovered or sitting idle topside after recovery.

18.4.1 Command Line Usage for the `alogclip` Tool

The `alogclip` tool is run from the command line with a given `.alog` file, a start time, end time, and the name of a new `.alog` file. By default, if the named output file exists, the user will be prompted before overwriting it. The user prompt can be bypassed with the `-f,--force` option. The usage options are listed when the tool is launched with the `-h` switch:

```
$ alogclip --help or -h
```

Listing 18.4: Command line usage for the `alogclip` tool.

```
1 Usage:  
2   alogclip in.alog mintime maxtime [out.alog] [OPTIONS]  
3  
4 Synopsis:  
5   Create a new MOOS .alog file from a given .alog file  
6   by removing entries outside a given time window.  
7  
8 Standard Arguments:  
9   in.alog - The input logfile.  
10  mintime - Log entries with timestamps below mintime  
11      will be excluded from the output file.  
12  maxtime - Log entries with timestamps above mintime
```

```

13           will be excluded from the output file.
14   out.alog - The newly generated output logfile. If no
15           file provided, output goes to stdout.
16
17 Options:
18   -h,--help      Display this usage/help message.
19   -v,--version   Display version information.
20   -f,--force     Overwrite an existing output file
21   -q,--quiet    Verbose report suppressed at conclusion.
22
23 Further Notes:
24   (1) The order of arguments may vary. The first alog
25       file is treated as the input file, and the first
26       numerical value is treated as the mintime.
27   (2) Two numerical values, in order, must be given.
28   (3) See also: alogscan, alogrm, aloggrep, alogview

```

18.4.2 Example Output from the `alogclip` Tool

The output shown below was generated from the `alpha.alog` file generated by the Alpha example mission.

```
$ alogclip alpha.alog new.alog 50 350
```

Listing 18.5: Example output for the `alogclip` tool.

```

1 Processing input file alpha.alog...
2
4 Total lines clipped:    44,988  (44.32 pct)
5   Front lines clipped:  5,474
6   Back  lines clipped: 39,514
7 Total chars clipped: 4,200,260  (43.09 pct)
8   Front chars clipped: 432,409
9   Back  chars clipped: 3,767,851

```

18.5 The `aloggrep` Tool

The `aloggrep` tool will prune a given `.alog` file by retaining lines of the original file that contain log entries for a user-specified list of MOOS variables or MOOS processes (sources). As the name implies it is motivated by the Unix `grep` command, but `grep` will return a matched line regardless of where the pattern appears in the line. Since MOOS variables also often appear in the string content of other MOOS variables, `grep` often returns much more than one is looking for. The `aloggrep` tool will only pattern-match on the second column of data (the MOOS variable name), or the third column of data (the MOOS source), of any given entry in a given `.alog` file.

18.5.1 Command Line Usage for the `aloggrep` Tool

```
$ aloggrep --help or -h
```

Listing 18.6: Command line usage for the `aloggrep` tool.

```
1 Usage:  
2   aloggrep in.alog [VAR] [SRC] [out.alog] [OPTIONS]  
3  
4 Synopsis:  
5   Create a new MOOS .alog file by retaining only the  
6   given MOOS variables or sources from a given .alog file.  
7  
8 Standard Arguments:  
9   in.alog - The input logfile.  
10  out.alog - The newly generated output logfile. If no  
11     file provided, output goes to stdout.  
12  VAR      - The name of a MOOS variable  
13  SRC      - The name of a MOOS process (source)  
14  
15 Options:  
16  -h,--help    Displays this help message  
17  -v,--version Displays the current release version  
18  -f,--force   Force overwrite of existing file  
19  -q,--quiet   Verbose report suppressed at conclusion  
20  
21 Further Notes:  
22  (1) The second alog is the output file. Otherwise the  
23      order of arguments is irrelevent.  
24  (2) VAR* matches any MOOS variable starting with VAR  
25  (3) See also: alogscan, alogrm, alogclip, alogview
```

Note that, in specifying items to be filtered out, there is no distinction made on the command line that a given item refers to a entry's variable name or an entry's source, i.e., MOOS process name.

18.5.2 Example Output from the `aloggrep` Tool

The output shown in Listing 7 was generated from the `alpha.alog` file generated by the Alpha example mission.

```
$ aloggrep alpha.alog NAV_* new.alog
```

Listing 18.7: Example `aloggrep` output applied to the `alpha.alog` file.

```

1 Processing on file : alpha.alog
2   Total lines retained: 54099 (53.30%)
3   Total lines excluded: 47396 (46.70%)
4   Total chars retained: 3293774 (33.79%)
5   Total chars excluded: 6453494 (66.21%)
6   Variables retained: (9) NAV_DEPTH, NAV_HEADING, NAV_LAT, NAV_LONG,
7   NAV_SPEED, NAV_STATE, NAV_X, NAV_Y, NAV_YAW

```

18.6 The `alogrm` Tool

The `alogrm` tool will prune a given `.alog` file by removing lines of the original file that contain log entries for a user-specified list of MOOS variables or MOOS processes (sources). It may be fairly viewed as the complement of the `aloggrep` tool.

18.6.1 Command Line Usage for the `alogrm` Tool

```
$ alogrm --help or -h
```

Listing 18.8: Command line usage for the `alogrm` tool.

```

1 $ alogrm -h
2
3 Usage:
4   alogrm in.alog [VAR] [SRC] [out.alog] [OPTIONS]
5
6 Synopsis:
7   Remove the entries matching the given MOOS variables or sources
8   from the given .alog file and generate a new .alog file.
9
10 Standard Arguments:
11   in.alog - The input logfile.
12   out.alog - The newly generated output logfile. If no
13     file provided, output goes to stdout.
14   VAR      - The name of a MOOS variable
15   SRC      - The name of a MOOS process (source)
16
17 Options:
18   -h,--help    Displays this help message
19   -v,--version Displays the current release version
20   -f,--force   Force overwrite of existing file
21   -q,--quiet   Verbose report suppressed at conclusion
22   --nostr     Remove lines with string data values
23   --nonum     Remove lines with double data values
24   --clean      Remove lines that have a timestamp that is
25     non-numerical or lines w/ no 4th column
26
27 Further Notes:
28   (1) The second alog is the output file. Otherwise the
29     order of arguments is irrelevant.
30   (2) VAR* matches any MOOS variable starting with VAR
31   (3) See also: alogscan, aloggrep, alogclip, alogview

```

Note that, in specifying items to be filtered out, there is no distinction made on the command line that a given item refers to a entry's variable name or an entry's source, i.e., MOOS process name.

18.6.2 Example Output from the `alogrm` Tool

The output shown in Listing 9 was generated from the `alpha.alog` file generated by the Alpha example mission.

Listing 18.9: Example `alogrm` output applied to the `alpha.alog` file.

```
1 $ alogrm alpha.alog NAV_* new.alog
2
3 Processing on file : alpha.alog
4
5 Total lines retained: 47396 (46.70%)
6 Total lines excluded: 54099 (53.30%)
7 Total chars retained: 6453494 (66.21%)
8 Total chars excluded: 3293774 (33.79%)
9 Variables retained: (48) BHV_IPF, CREATE_CPU, CYCLE_INDEX, DB_CLIENTS,
10 DB_TIME, DB_UPTIME, DEPLOY, DESIRED_HEADING, DESIRED_RUDDER, DESIRED_SPEED,
11 DESIRED_THRUST, HELM_IPF_COUNT, HELM_MAP_CLEAR, HSLINE, USIMMARINE_STATUS,
12 IVPHELM_DOMAIN, IVPHELM_ENGAGED, IVPHELM_MODESET, IVPHELM_POSTINGS,
13 IVPHELM_STATEVARS, IVPHELM_SUMMARY, LOGGER_DIRECTORY, LOOP_CPU, MOOS_DEBUG,
14 MOOS_MANUAL_OVERRIDE, NODE_REPORT_LOCAL, PC_hslide, PC_waypt_return,
15 PC_waypt_survey, PHELMIVP_STATUS, PLOGGER_CMD, PLOGGER_STATUS,
16 PMARINEPID_STATUS, PMARINEVIEWER_STATUS, PNODEREPORTER_STATUS,
17 PWT_BHV_HSLINE, PWT_BHV_WAYPT_RETURN, PWT_BHV_WAYPT_SURVEY, RETURN,
18 STATE_BHV_HSLINE, STATE_BHV_WAYPT_RETURN, STATE_BHV_WAYPT_SURVEY,
19 SURVEY_INDEX, SURVEY_STATUS, VIEW_POINT, VIEW_SEGLIST, WPT_INDEX, WPT_STAT
```

18.7 The `alogview` Tool

The `alogview` application is used for post-mission rendering of one or more alog files. It provides (a) an indexed view of vehicle position rendered on the operation area, (b) time plots of any logged numerical data, (c) IvP Helm information for a given point in time, and (d) rendered IvP functions generated by the helm for a given point in time. A snapshot of the tool is shown in Figure 72.

This tool is very much still under development, and the below documentation is far from complete despite the best intentions after release 4.1. Nevertheless, since there are those who are using this regularly at this date, some attempt here is made to introduce the tool. This tool was also known as `logview` prior to release 4.1 in August 2010. It was renamed to `alogview` to make it consistent with other tools in the Alog Toolbox. A significant addition to the tool since Release 4.1 is the support for rendering depth objective functions as shown in the figure.

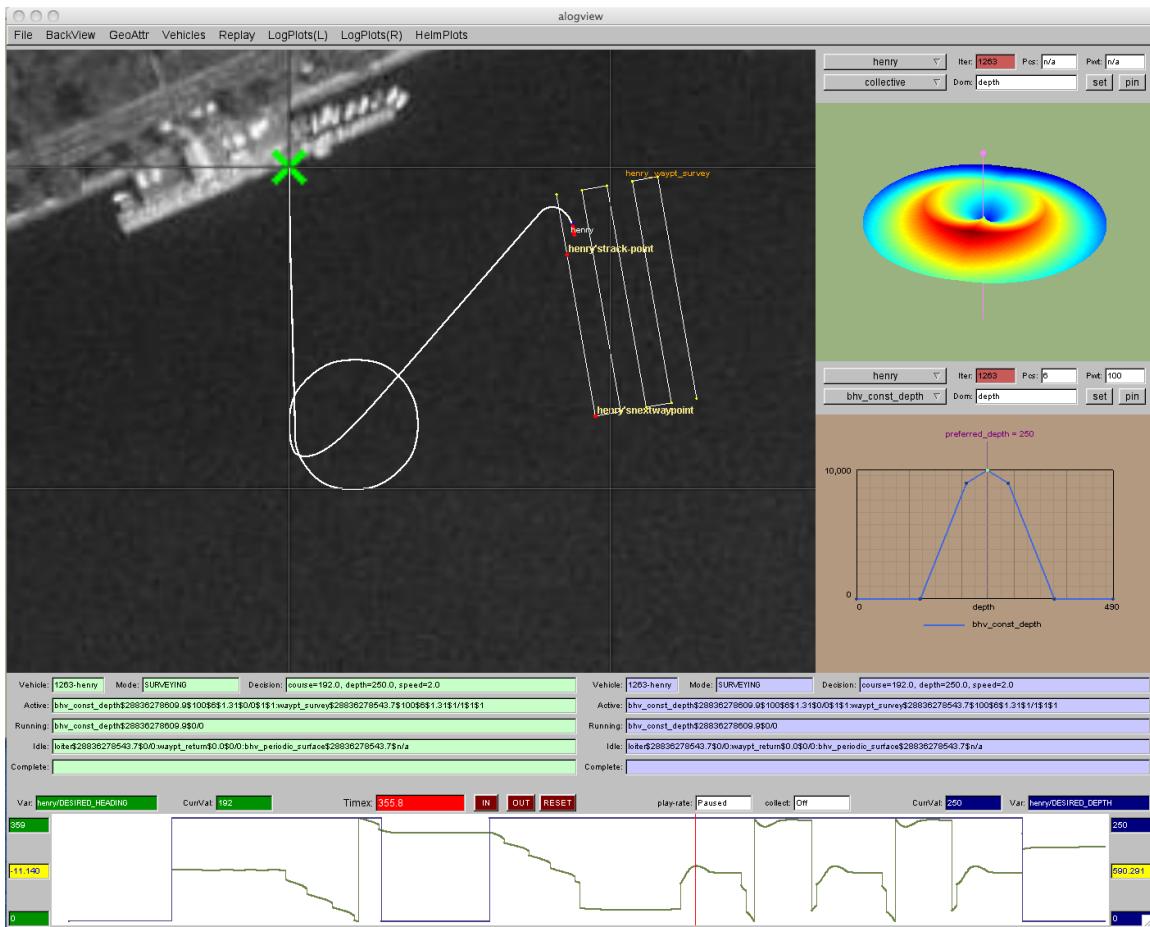


Figure 72: **The alogview tool:** used for post-mission rendering of alog files from one or more vehicles and stepping through time to analyze helm status, IvP objective functions, and other logged numerical data correlated to vehicle position in the op-area and time.

The view shown to the user at any given time is indexed on a timestamp. The vehicles rendered in

the op-area are shown at their positions for that point in time. The helm scope output and IvP function output are displayed for the helm iteration at the current timestamp. The data plot output shows a plot for a given variable over all logged time with a red vertical bar that moves left to right indicating the current time. The `alogview` tool by default loads the complete alog vehicle history to allow the user to jump directly to any point in time. The option also exists to load only a portion of the data based on start and end time provided on the command line.

18.7.1 Command Line Usage for the `alogview` Tool

The `alogview` tool is run from the command line with one or more given `.alog` files and a number of options. The usage options are listed when the tool is launched with the `-h` switch:

```
$ alogview --help or -h
```

Listing 18.10: Command line usage for the `alogview` tool.

```
1 Usage:
2   alogview file.alog [another_file.alog] [OPTIONS]
3
4 Synopsis:
5   Renders vehicle paths from multiple MOOS .alog files.
6   Renders time-series plots for any logged numerical data.
7   Renders IvP Helm mode information vs. vehicle position.
8   Renders IvP Helm behavior objective functions.
9
10 Standard Arguments:
11   file.alog - The input logfile.
12
13 Options:
14   -h,--help      Displays this help message
15   -v,--version   Displays the current release version
16   --mintime=val  Clip data with timestamps < val
17   --maxtime=val  Clip data with timestamps > val
18   --nowtime=val   Set the initial startup time
19   --geometry=val  Viewer window pixel size in HEIGHTxWIDTH
20           or large, medium, small, xsmall
21           Default size is 1400x1100 (large)
22   --layout=val    Window layout=normal,noipfs, or fullview
23
24 Further Notes:
25   (1) Multiple .alog files ok - typically one per vehicle
26   (2) Non alog files will be scanned for polygons
27   (3) See also: alogscan, alogrm, alogclip, aloggrep
```

The order of the arguments passed to `alogview` do not matter. The `--mintime` and `--maxtime` arguments allow the user to effectively clip the alog files to reduce the amount of data loaded into RAM by `alogview` during a session. The `--geometry` argument allows the user to custom set the size of the display window. A few shortcuts, "large", "medium", "small", and "xsmall" are allowed. The `--layout` argument allows the user to affect the real estate layout by optionally closing one or more panels to enlarge other panels. This is described in Section 18.7.2.

18.7.2 Description of Panels in the `alogview` Window

Although the `alogview` tool will read in any `.alog` file produced by the `pLogger` tool, much of the tool's screen real estate is dedicated to rendering information produced by the helm. The `alogview` tool has six panels of information, as shown in Figure 73. The primary panel is the Op-Area Panel which renders the vehicle position(s) on the operation area as a function of time, along with track history. The IvPFunction Panels render the objective functions produced by vehicle behaviors for a given helm iteration. The Helm Scope Panels display helm output and behavior information for a given helm iteration. The Data Plot panels render two plots of logged numerical data vs the vehicle log time. Each of these panels and panel controls are discussed in the next few sections.

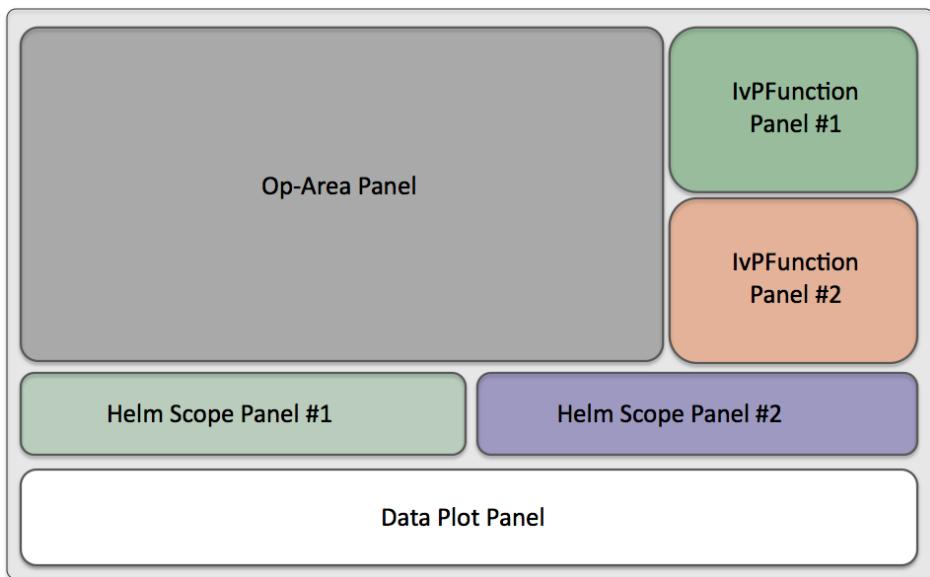


Figure 73: **The panels comprising the `alogview` tool:** Each panel renders information based on the globally held current timestamp. Certain panels may be collapsed to make more room for other panels.

18.7.3 The Op-Area Panel for Rendering Vehicle Trajectories

The Op-Area panel renders, for a given point in time, the vehicle(s) current position and orientation, the vehicle(s) trajectory history, and certain geometric objects such as points, polygons, line segments and their labels, that may have been posted by the helm or other MOOS processes if they were logged in the `alog` file(s). An example is shown in Figure 74.

Stepping Through Time - The Playback Pull-down Menu

The primary interaction with the Op-Area panel is to step the vehicle forward and backward through time either by fixed time increments or by helm iteration. The simplest way to step is with either the '[' and ']' keys to step backward and forward by one step, or the '<' and '>' keys to move by ten steps. (The current time can also be jumped to with a mouse click in the Data Plot panel. A

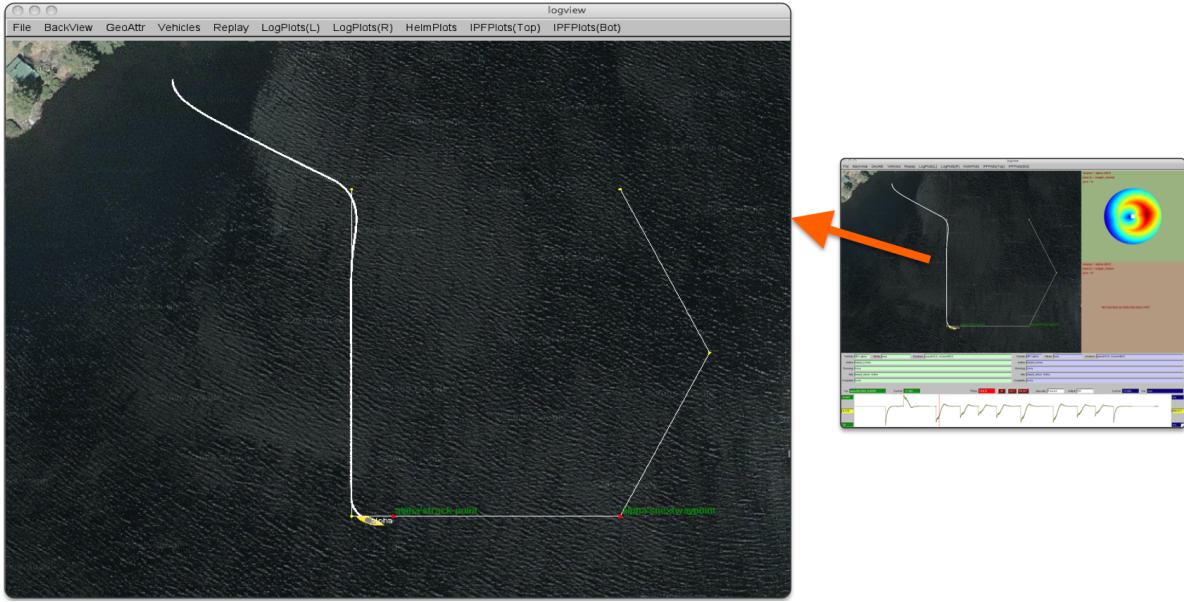


Figure 74: **The OpArea panel of the `alogview` tool:** Vehicle position(s) for a given point in time are rendered along with vehicle trajectory history and certain geometric visual artifacts such as points, polygons and line segments that may have been posted by the helm or another MOOS process. The image in this figure can be replicated exactly by launching `alogview` on the `alpha.alog` file from the Alpha mission distributed with the MOOS-IvP source code, launched with the `--nowtime=144` command line option.

step unit by default is one second. Alternatively the step unit may be given by one iteration of the helm. If multiple vehicles (alog files) are open, a helm iteration is defined by the helm in the “active” vehicle, i.e., the vehicle whose helm scope information is being displayed in the left-hand Helm Scope panel.

The stepping can be initiated by successive key clicks as noted above or be automatic when put into playback, i.e., streaming mode. In streaming mode the steps continue automatically at fixed intervals until paused or until the end of the latest timestamp of all loaded log files. The time interval between step executions can be sped up or slowed down with the ‘a’ or ‘z’ keys respectively. The primary motivation of streaming was to have the option of doing a screen capture of images on each step saved to a file for later compilation as an animation (typically animated GIF). Screen capturing can be enabled from the Playback pull-down menu or by hitting the ‘w’ key. When enabled, a purple box should be rendered over the Op-Area panel indicating the scope of the screen capture. By successively hitting the ‘w’ key, the capture box is changed between the following extents: "1024x768", "800x600", "640x480", "480x360", and "off". The capturing is done by invoking a system call to the `import` tool distributed with the powerful ImageMagick Open Source package.

Vehicle and Vehicle History Renderings - The Vehicles Pull-down Menu

The rendering of vehicle size, type and color, vehicle trails and the position of the vehicles may be altered via the Vehicles pull-down menu options. The vehicle size and shape upon startup is determined from the `NODE_REPORT_LOCAL` variable (See Section 4.2.1). The set of displayable shapes

in `alogview` is the same as that for the `pMarineViewer` application and shown in Figure 23. The rendering of vehicles in the Op-Area panel may be toggled off and on by hitting the `CTRL-'v'` key, and the size of the vehicles may be altered with the `-` and `+` keys. The size of the rendered vehicle initially is drawn to scale based on the length reported in `NODE_REPORT_LOCAL`, and can be returned to scale by hitting the `ALT-'v'` key.

The rendering of vehicle names may be toggled off and on by hitting the `'n'` key, and the user may toggle between a few different choices for text color by hitting the `ALT-'n'` key. By default the color of the vehicles is set to be yellow for all *inactive* vehicles, and red for the one *active* vehicle (where *active* means it is the vehicle whose helm data is shown in the left-hand Helm Scope panel). A few different choices for active and inactive vehicle colors are provided in the Vehicles pull-down menu. The selection of the active vehicle can be made explicitly from the HelmPlots pull-down menu, by cycling through the vehicles by hitting the `'v'` key.

Vehicle trails, i.e., position history, are by default rendered from the present point in time back to the beginning of logged positions in the alog file. Three other modes are supported and can be toggled through with the `'t'` key. The other modes are: no trails shown at all, all trails shown from the start to end log time, and trails with a limited history. In the latter case the trail stays a fixed length behind the vehicle. This fixed length can be made shorter or longer with the `'('` or `')'` keys respectively. The size of each trail point can be made smaller or larger with the `'['` or `']'` keys respectively.

Geometric Object Renderings - The GeoAttr Pull-down Menu

Certain geometric objects logged in the alog file may be displayed in the Op-Area panel and their renderings affected by choices available in the GeoAttr pull-down menu. Each object is posted with a tag that allows for the object to be effectively erased when an object of the same type and tag is subsequently posted. The `alogview` tool, upon startup, reads through the log file(s) and determines which geometric objects are viewable at any given point in time. Thus the user may see these objects disappear and reappear as one steps back and forth through time. Current objects supported by `alogview` are `VIEW_POINT`, `VIEW_SEGLIST`, `VIEW_POLYGON`, `VIEW_MARKER`, and `VIEW_RANGE_PULSE`

18.7.4 The Helm Scope Panels for View Helm State by Iteration

The *helm scope* panels are used for examining the state of the vehicle helm at the present point in time. In the `Vehicle:` box, the name of the vehicle preceded by the helm iteration is shown. In the `Mode:` box, the helm's current mode is given if hierarchical mode declarations are configured for the helm. In the `Decision:` box, the helm's decision for that iteration is shown for each decision variable. The `Active:` box, shows the list of behaviors active on the present helm iteration. The `Running:` box, shows the list of behaviors running on the present helm iteration. The `Idle:` box, shows the list of behaviors idle on the present helm iteration. The `Completed:` box, shows the list of behaviors that have been completed as of the present helm iteration.

If there are multiple vehicles (alog files) being viewed, the user can switch the vehicle for each helm scope panel via the HelmPlots pull-down menu.

18.7.5 The Data Plot Panel for Logged Data over Time

The *data plot* panels at the bottom of the window allow the user to plot any two logged MOOS variables, if they were logged as numerical data. A red bar in the time plot indicates the current point in time so the user can visually correlate the vehicles' position in the op area relative to the data plot. The user can also click anywhere on the time plot to alter the current point in time used in all panels. The user may also zoom in on the data plot for better resolution. One note of caution - the scales used by the two variables are likely not the same. The range from low to high for the particular variable. The range of values is shown on the far left and far right.

18.7.6 Automatic Replay of the Log file(s)

The user can allow the `alogview` tool to step through time automatically, effectively replaying the mission over its duration. Replaying can be adjusted in the Replay pull-down menu. The '=' key toggles replaying, and the 'a' and 'z' keys can be used to slow down or speed up the replay rate. This feature is useful when used with other software that allows automatic generation of video capture real-time from the display. QuickTime in OS X for example.

19 uFldNodeBroker: Brokering Node Connections

The `uFldNodeBroker` application is a tool for brokering connections between a node (a simulated or real vehicle) and a shoreside community. It is used primarily in coordination with `uFldShoreBroker` to discover and share host IP and port information to automate the dynamic configurations of `pShare`. Inter-vehicle communications over the network are handled by `pShare` in both simulation with single or multiple machines as well as on fielded vehicles using Wi-Fi or cellphone connections. The `pShare` application simply needs to know the IP address and port number of connected machines. Often these aren't known at run-time and even if they were, maintaining that information in configuration files may be unduly cumbersome, especially for large sets of vehicles. This tool meant to automate the configuration by letting the nodes and shoreside community discover each other by giving the node (`uFldNodeBroker`) some initial hints on where to find the shoreside community on the network. The typical layout is shown in Figure 75.

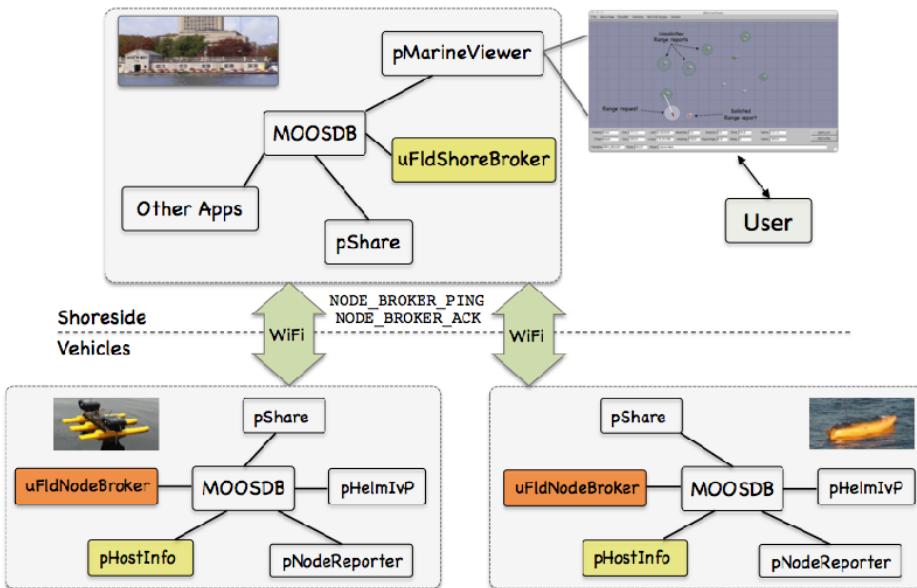


Figure 75: **Typical uFldNodeBroker Topology:** A vehicle (node) sends information about itself (IP address and port number) to possible shoreside locations. Once a connection is made, further bridging is established between the node and shoreside communities.

The functionality of `uFldNodeBroker` paraphrased:

- Discover the node's host information (typically from `pHostInfo`).
- For candidate shoreside hosts, request a new bridge in `pShare` to each candidate for the variable `NODE_BROKER_PING`.
- Publish `NODE_BROKER_PING` with the node's host information.
- Await a reply in the form of incoming `NODE_BROKER_ACK` mail, presumably from the shoreside community running `uFldShoreBroker`.
- Now that a shoreside community is known, request new bridges from the node's local `pShare` for all the info we otherwise want bridged to the shoreside.

- Keep sending pings periodically in case the shoreside community is re-started and needs to re-establish connections to nodes.

19.1 The uFldNodeBroker Interface and Configuration Options

The `uFldNodeBroker` application may be configured with a configuration block within a `.moos` file. Its interface is defined by its publications and subscriptions for MOOS variables consumed and generated by other MOOS applications. An overview of the set of configuration options and interface is provided in this section. If one has access to a command line where `uFldNodeBroker` has been built, interface information may also be seen by typing "`uFldNodeBroker --interface`", and configuration information by typing "`uFldNodeBroker --example`".

19.1.1 Configuration Parameters of uFldNodeBroker

The following parameters are defined for `uFldNodeBroker`.

Listing 19.1: Configuration Parameters for `uFldNodeBroker`.

- `keyword`: An optional unique identifier to be read by the shoreside broker to conditionally respond to pings.
- `bridge`: A variable to register with `pShare` for bridging to a shoreside MOOS community once a shoreside connection has been established.
- `try_shore_host`: A candidate route to send initial pings in hopes of establishing a connection. The route specifies an input route for `pShare` running in a shoreside community.

An Example MOOS Configuration Block

An example MOOS configuration block can be obtained by entering the following from the command-line:

```
$ uFldNodeBroker --example or -e
```

Listing 19.2: Example configuration of the `uFldNodeBroker` application.

```

1 =====
2 uFldNodeBroker Example MOOS Configuration
3 =====
4
5 ProcessConfig = uFldNodeBroker
6 {
7   AppTick    = 4
8   CommsTick = 4
9

```

```

10 keyword      = lemon
11
12 try_shore_host = pshare_route=localhost:9200
13 try_shore_host = pshare_route=192.168.0.122:9301
14 try_shore_host = pshare_route=multicast_8
15
16 bridge = src=VIEW_POLYGON
17 bridge = src=VIEW_POINT
18 bridge = src=VIEW_SEGLIST
19
20 bridge = src=NODE_REPORT_LOCAL, alias=NODE_REPORT
21 }

```

19.2 Publications and Subscriptions for uFldNodeBroker

The interface for `uFldNodeBroker`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldNodeBroker --interface or -i
```

19.2.1 Variables Published by uFldNodeBroker

The primary output of `uFldNodeBroker` to the MOOSDB are the requests to `pShare` for registrations, and the outgoing pings to candidate shoreside communities.

- **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 19.3.
- **NODE_BROKER_PING**: A message written locally but bridged to a candidate shoreside MOOS community, containing IP address and `pShare` route information about the local community.
- **PSHARE_CMD**: message to `pShare` to add a new bridge for a given variable and given target MOOS community at a specified IP address and port number.

19.2.2 Variables Subscribed for by uFldNodeBroker

The `uFldNodeBroker` application subscribes to the following MOOS variables:

- **APPCAST_REQ**: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 11.10.4.
- **NODE_BROKER_ACK**: Information published presumably by `uFldShoreBroker` running in a separate shoreside community. Message has information about the shoreside host including the community name, IP address and port numbers for the MOOSDB and its local `pShare` process.
- **PHI_HOST_INFO**: Information about the local host IP address, the MOOS community name, the port on which the DB is running, and the port on which the local `pShare` is listening for UDP messages.

19.2.3 Command Line Usage of uFldNodeBroker

The `uFldNodeBroker` application is typically launched with `pAntler`, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing

```
$ uFldNodeBroker --help or -h
```

Listing 19.3: Command line usage for the `uFldNodeBroker` tool.

```
1 =====
2 Usage: uFldNodeBroker file.moos [OPTIONS]
3 =====
4
5 Options:
6   --alias=<ProcessName>
7     Launch uFldNodeBroker with the given
8     process name rather than uFldNodeBroker.
9   --example, -e
10    Display example MOOS configuration block.
11   --help, -h
12    Display this help message.
13   --interface, -i
14    Display MOOS publications and subscriptions.
15   --version,-v
16    Display release version of uFldNodeBroker.
```

19.3 Terminal and AppCast Output

The `uFldNodeBroker` application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 4 below. This application is also appcast enabled, meaning its reports are published to the MOOSDB and viewable from any uMAC application or pMarineViewer. See Section 10 for more on appcasting and viewing appcasts. The counter on the end of line 2 is incremented on each iteration of `uFldNodeBroker`, and serves a bit as a heartbeat indicator. The "0/0" also on line 2 indicates there are no configuration or run warnings detected.

Listing 19.4: Example terminal or appcast output for `uFldNodeBroker`.

```
1 =====
2 uFldNodeBroker henry          0/0(129)
3 =====
4
5   Total OK  PHI_HOST_INFO    received: 13
6   Total BAD PHI_HOST_INFO    received: 0
7   Total HOST_INFO changes   received: 1
8   Total          PSHARE_CMD  posted: 7
9   Total BAD NODE_BROKER_ACK received: 6
10
11 =====
```

```

12          Vehicle Node Information:
13 =====
14
15      Community: henry
16          HostIP: 10.0.0.5
17      Port MOOSDB: 9001
18      Time Warp: 4
19      IRoutes: 10.0.0.5:9301
20
21 =====
22          Shoreside Node(s) Information:
23 =====6
24
25 Community          Pings  Pings  IP       Time
26 Name    Route        Sent    Acked   Address  Warp
27 -----
28 shoreside localhost:9300 128     120    10.0.0.5 4
29 Phase Completion Summary:
30 -----
31 Phase 1: (Y) Valid Host information retrieved (iroutes).
32 Phase 2: (Y) Valid TryHosts (1) configured.
33 Phase 3: (Y) NODE_BROKER_PINGS are being sent to TryHosts.
34 Phase 4: (Y) A Valid NODE_BROKER_ACK has been received.
35 Phase 5: (Y) pShare requested to share user vars with shoreside.
36 All Phases complete. Things should be working as configured.

```

On line 5, the number of incoming mail messages for `PHI_HOST_INFO` is tallied where the host information bundle is deemed complete. It is complete if it contains the host community, IP address, time warp and `pShare` input route information. If an incomplete host information packet is received, it is tallied in line 6. The number on line 6 should always be zero. If the number on line 6 is not zero, or the number on line 5 never increments, you should check the operation of the `pHostInfo`.

The number on line 7 indicates the number of time the detected host information changes. This number should stabilize very quickly maxing out at 1 or 2 typically. The number on line 8 indicates the number of dynamic bridge requests posted to `pShare`. These are in the form of postings to the variable `PSHARE_CMD`, which occur first for outgoing pings to candidate shoreside hosts, and then for the user `bridge` configuration variables after a shoreside host has been connected. Invalid `NODE_BROKER_ACK` messages are tallied on line 9. An invalid ack may be due to a mismatch in time warp between node and shoreside, or a mismatch in keywords if keywords are being used.

Self node information is displayed in the next group, lines 11-19. The community name, MOOSDB port, and time warp are all read from the .moos mission configuration file. The node's IP address (line 16) and the local `pShare` input routes (line 19) are obtained from output received from `pHostInfo`.

Information about candidate and connected shoreside communities is shown next in lines 21-28. In this case there is only one entry, line 27. For each candidate entry, the community name, shoreside `pShare` input route, number of pings sent and acknowledged are in the first four columns. The last two columns indicate the IP address and time warp of the shoreside learned from `NODE_BROKER_ACK` messages received from the shoreside. When a candidate shoreside community has *not* been connected, the entry in this table will something like:

Community	Pings	Pings	IP	Time
-----------	-------	-------	----	------

Name	Route	Sent	Acked	Address	Warp
localhost:92003	385	0			

The last block of information in the report is the Phase Completion Summary, lines 29-36. It lists the rough sequence of events typical to reach a shoreside community connection. If any one of these phases is incomplete, the output on line 36 will be replaced with a few hints on where to troubleshoot.

20 uFldShoreBroker: Brokering Shore Connections

The `uFldShoreBroker` application is a tool for brokering connections between a shoreside community and one or more nodes (simulated or real vehicles). A shoreside community is collection of MOOS processes typically running a GUI providing a situational display and managing messages to and from fielded vehicles. This is depicted in the notional rendering in Figure 76 below. The shoreside community in practice is often situated on a ship with UUVs below, and is more aptly referred to as the topside community. The user interacts with the GUI or perhaps other communication modules, to send high-level messages to the vehicles.

The `uFldShoreBroker` application is used primarily in coordination with `uFldNodeBroker`, running on the vehicles, to discover and share host IP and port information to automate the dynamic configurations of `pShare`. Inter-vehicle communications over the network are handled by `pShare` in both simulation with single or multiple machines as well as on fielded vehicles using Wi-Fi or cellphone connections. The `pShare` application simply needs to know the IP address and port number of connected machines. Often these aren't known at run-time and even if they were, maintaining that information in configuration files may be unduly cumbersome, especially for large sets of vehicles. This tool is meant to automate the configuration by letting the nodes and shoreside community discover each other by letting (`uFldShoreBroker`) respond to incoming pings, i.e., initialization messages, from nodes on the network. The typical layout is shown in Figure 75.

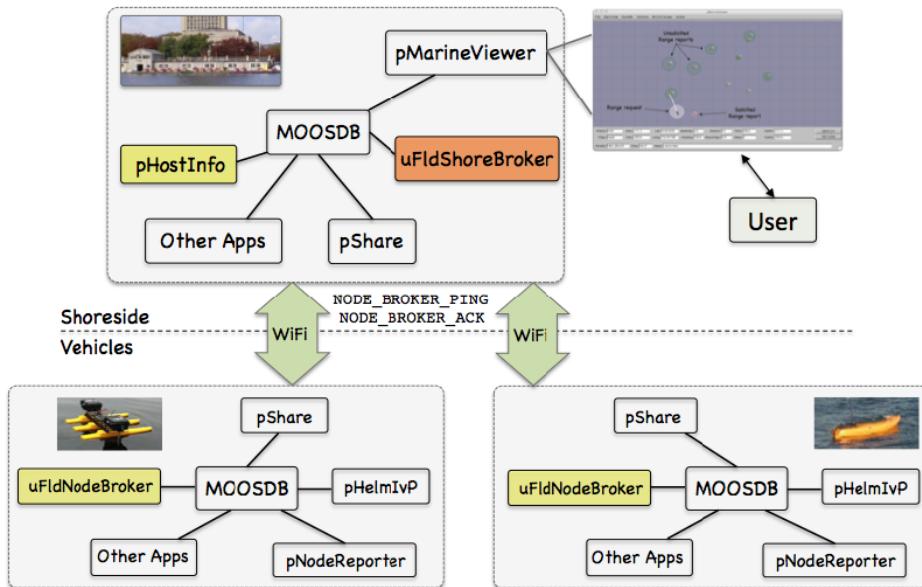


Figure 76: **Typical uFldShoreBroker Topology:** A vehicle (node) sends information about itself (IP address and port number) to the shoreside, received by `uFldShoreBroker`. It responds by (a) acknowledging the connection to the node, and (b) establishing user configured bridges of particular MOOS variables to the node.

The functionality of `uFldShoreBroker` paraphrased:

- Discover the shoreside's own host information (typically from `pHostInfo`).
- Await incoming `NODE_BROKER_PING` messages from non-local vehicles.

- Upon an incoming ping, respond to the nodes with a `NODE_BROKER_ACK` message to the location specified in the ping message.
- Establish new bridges to the nodes for variables specified previously by the user in the `uF1dShoreBroker` configuration.
- Keep sending acknowledgements periodically to confirm to vehicles that they are still connected to the shoreside community.

20.1 Bridging Variables Upon Connection to Nodes

A primary function of `uF1dShoreBroker` is to establish bridging relationships to a remote node community after it has received a ping from that community. These variables are specified in the configuration block with lines like `bridge = "src=DEPLOY_ALL, alias=DEPLOY"` as in Listing 3. This step is described next.

20.1.1 Inter-MOOSDB Bridging with pShare

Static bridging with `pShare` is done by specifying the desired route in the `pShare` configuration block with a line of the form:

```
output = src_name=VAR, dest_name=ALIAS, route=ROUTE
```

For example:

```
output = src_name=DEPLOY_HENRY, dest_name=DEPLOY, route=12.56.111.1:9200
```

The above connection may be used to send the vehicle Henry the deploy command from the shoreside community. The problem is that the shoreside may not know the IP address of Henry (or the port on which its `pShare` is listening) until it presents itself to the shoreside at run time.

In this case, dynamic share registration needs to be used by sending `pShare` a message after it has been launched. For example, the above sharing relationship could be established by sending the following message:

```
PSHARE_CMD = "cmd=output, src_name=DEPLOY_ALL, dest_name=DEPLOY,
route=2.56.111.1:9200
```

It is the job of `uF1dShoreBroker` to post the above style dynamic requests once the node information becomes known to the shoreside community.

20.1.2 Handling a Valid Incoming Ping from a Remote Node

The basic job of `uF1dShoreBroker` is to await incoming pings, and use the information in a ping message to (a) decide if the ping should be accepted, and (b) send the appropriate response back to the sender, and (c) set up new outgoing `pShare` relationships if the ping is indeed accepted. The contents of a ping may look something like:

```
NODE_BROKER_PING = "community=henry,host=192.168.1.22,port=9000,time_warp=10  
pshare_iroutes=192.168.1.22:9200,time=1325178800.81"
```

There must be a match in the MOOS *time warp* used by the shoreside MOOS community and any node connected to the shore. This is always 1 when operating vehicles in the field, but may be set to a much larger number in simulation. The time warp is set with the parameter `MOOSTimeWarp` at the top of the .moos configuration file.

The ping consists of three key pieces of information:

- The community name of the node,
- The IP address of the node,
- The input routes on which the node is listening for messages with its own local `pShare` running.

Once this information is known by the shoreside broker, new bridges can be established for variables identified by the user. The only other information needed is (a) the name of the variable in the local `MOOSDB`, and (b) the name (alias) of the variable as it is to be known in the remote `MOOSDB`. Once a valid ping has been received and accepted, `uFldShoreBroker` is ready to establish bridge arrangements with its local `pShare` running.

20.1.3 Vanilla Bridge Arrangements

The simplest bridge arrangement specifies (a) the variable as it is known locally, and (b) the variable name as it is to be known remotely. This is done with a `uFldShoreBroker` configuration line similar to:

```
bridge = src=DEPLOY_ALL, alias=DEPLOY
```

For *each* unique incoming ping, a new bridge arrangement will be requested. By unique, we mean having a distinct community (remote vehicle) name. For example, if pings are received and accepted from *henry*, *james*, and *ike*, `uFldShoreBroker` would make three separate posts, perhaps looking like:

```
PSHARE_CMD = "src_name=DEPLOY_ALL, dest_name=DEPLOY, route=2.56.111.1:9200"  
PSHARE_CMD = "src_name=DEPLOY_ALL, dest_name=DEPLOY, route=2.56.111.3:9200"  
PSHARE_CMD = "src_name=DEPLOY_ALL, dest_name=DEPLOY, route=2.56.111.6:9200"
```

At this point the behavior of `pShare` on the shoreside would be functionally equivalent to the scenario where the following three lines were in the `pShare` configuration block:

```
output = src_name=DEPLOY_ALL, dest_name=DEPLOY, route=12.56.111.1:9200  
output = src_name=DEPLOY_ALL, dest_name=DEPLOY, route=12.56.111.3:9200  
output = src_name=DEPLOY_ALL, dest_name=DEPLOY, route=12.56.111.6:9200
```

20.1.4 Bridge Arrangements with Macros

The user may configure `uF1dShoreBroker` with bridge arrangements containing a couple types of macros. For example, consider the configuration:

```
bridge = src=DEPLOY_$V, alias=DEPLOY
```

The `$V` macro will expand to the name of the vehicle when it comes time to request a new bridge. If the newly received ping is from the node named *gilda*, the bridge request from the above pattern may look like:

```
PSHARE_CMD = cmd=output, src_name=DEPLOY_GILDA, dest_name=DEPLOY,  
route=2.56.111.1:9200
```

Note the vehicle name in the MOOS variable macro was expanded to be upper case, even though the ping information referred to the vehicle as *gilda*. This is just to aid in the convention that MOOS variable are typically all upper case. If one really want a literal expansion with no case altering, the macro `$v`, lower-case `v`, may be used instead. The macro is only respected as part of `src` field. In other words, if the bridge were configured with `alias=DEPLOY_$V`, the macro would not be expanded.

The other type of macro implemented is the `$N` macro, as in:

```
bridge = src=LOITER_$N, alias=LOITER
```

The `$N` macro will expand to the integer value representing number of unique pings received thus far. For example, if three pings are received and accepted from *henry*, *james*, and *ike*, `uF1dShoreBroker` would make three separate posts, perhaps looking like:

```
PSHARE_CMD = "src_name=LOITER_1, dest_name=LOITER, route=2.56.111.1:9200"  
PSHARE_CMD = "src_name=LOITER_2, dest_name=LOITER, route=2.56.111.4:9200"  
PSHARE_CMD = "src_name=LOITER_2, dest_name=LOITER, route=2.56.111.12:9200"
```

This may be useful when used in conjunction with another MOOS process generating output generically for N vehicles, without having to know the vehicle names in advance.

20.1.5 A Common Configuration Shortcut - the qbridge Parameter

A common usage pattern is to configure `uF1dShoreBroker` to request two types of bridges for a given variable, for example:

```
bridge = src=DEPLOY_ALL, alias=DEPLOY  
bridge = src=DEPLOY_$V, alias=DEPLOY  
bridge = src=RETURN_ALL, alias=RETURN  
bridge = src=RETURN_$V, alias=RETURN
```

This could be used in the shoreside community for easily commanding vehicles. When the user wishes to deploy all vehicles, a posting of `DEPLOY_ALL="true"` does the trick. If the user wishes only the vehicle *james* to return, a posting of `RETURN_JAMES="true"` may be made. This pattern is so common that this shortcut is supported. This is done with the `qbridge`, "quick bridge", parameter. The above four configuration lines could be accomplished instead by:

```
qbridge = DEPLOY, RETURN
```

20.2 Usage Scenarios for the uFldShoreBroker Utility

The `uFldShoreBroker` was designed with a canonical command-and control scenario in mind. The idea is that the N deployed vehicles have a common autonomy protocol implemented. For example, a message to deploy or return a vehicle is the same message for each deployed vehicle. The idea is that two types of communication channels need to be established with `pShare`, (a) messages sent to all vehicles, and (b) messages sent to a particular named vehicle. The convention proposed here is to do this with the two types of bridging described in the discussion of the `qbridge` parameter, in Section 20.1.5. For a variable such as `DEPLOY`, a posting in the shoreside community to `DEPLOY_ALL` would go to all known vehicles, and a posting to `DEPLOY_HENRY` would only go to that particular vehicle.

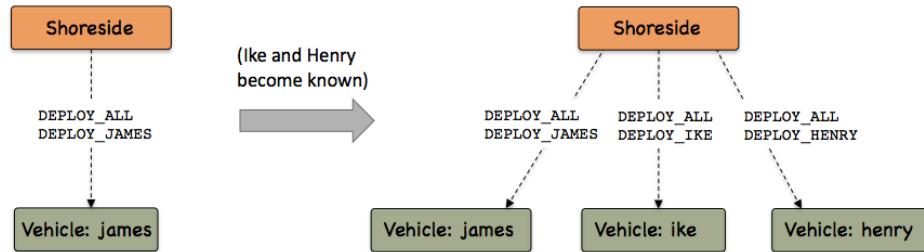


Figure 77: **Common uFldShoreBroker Usage Scenario:** As vehicles become known to the shoreside, each vehicle has two new bridges established. The first is the same for all vehicles to allow broadcasting, and the second bridge is unique to the particular vehicle, for individual command and control.

20.3 Terminal and AppCast Output

The `uFldShoreBroker` application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 1 below. This application is also appcast enabled, meaning its reports are published to the `MOOSDB` and viewable from any `uMAC` application or `pMarineViewer`. See Section 10 for more on appcasting and viewing appcasts.

On line 1, the application iteration is shown, more as a heartbeat indicator. In lines 4-7, the primary variables consumed and posted by `uFldShoreBroker` are summarized in terms of how many posts have been made and received for each variable.

Listing 20.1: Example terminal output of the uFldShoreBroker tool.

```
1 =====
```

```

2 uFldShoreBroker_PS shoreside                                (109)
3 =====
4
5   Total PHI_HOST_INFO    received: 11
6   Total NODE_BROKER_PING received: 180
7   Total NODE_BROKER_ACK     posted: 180
8   Total PSHARE_CMD        posted: 34
9
10
11 =====
12           Shoreside Node(s) Information:
13 =====
14
15       Community: shoreside
16             HostIP: 128.30.27.202
17             Port MOOSDB: 9000
18             Time Warp: 6
19             IRoutes: localhost:9200
20
21 =====
22           Vehicle Node Information:
23 =====
24
25 Node   IP                               Elap  pShare
26 Name   Address      Status  Time   Input Route(s)      Skew
27 -----  -----
28 henry  128.30.27.202  ok     0.0    128.30.27.202:9301  1.6542
29 gilda  128.30.27.202  ok     0.0    128.30.27.202:9302  1.7572
30
31 Recent Events (2):
32 [22.06]: New node discovered: gilda
33 [16.04]: New node discovered: henry

```

In lines 11-19, the key shoreside properties are listed. Typically, but not always, the shoreside community is name "shoreside" as indicated on line 15. The shoreside IP address, determined by `pHostInfo`, is shown on line 16. The time warp, and `MOOSDB` port are read from the shoreside .moos file and listed on lines 17 and 18. The input routes used by `pShare` are listed on line 19. If lines 16 or 19 are blank, `uFldShoreBroker` will not make any connections and the first place to look is whether or not `pHostInfo` is running and producing valid information.

In lines 21-33, the status of each of the known vehicles is shown. The first two vehicles had their pings accepted. Their IP addresses are shown in the second column. Their status is shown in the third column. The elapsed time in the fourth column is the time since the last ping was received by the shoreside. The fifth column shows the input routes being used by `pShare` running on the vehicle node. If multiple routes are in use, this will be shown over multiple lines. The sixth column shows the time skew between the timestamp in the `NODE_BROKER_PING` message compared to the time it was received. Some of this is due to (a) latency in transmission, (b) latency due to App Ticks in brokers on both sides, and (c) clock discrepancy between the shoreside and the node computers. It's also worth mentioning that the skew will be magnified for higher time warps. Currently incoming ping connection requests are not denied due to a high clock skew, but this may be implemented in the future.

20.4 Configuration Parameters of uFldShoreBroker

The following parameters are defined for `uFldShoreBroker`.

Listing 20.2: Configuration Parameters for `uFldShoreBroker`.

`bridge`: Names a MOOS variable to be bridged to a node community.

`qbridge`: Shorthand notation for a common bridging pattern.

As an example, `bridge = src =DEPLOY_ALL, alias=DEPLOY`, will result in the bridging of variable `DEPLOY_ALL` from the local `MOOSDB`, to the variable `DEPLOY` in a remote MOOS community. Further examples are given in Section 20.1.

An Example MOOS Configuration Block

Listing 3 shows an example MOOS configuration block produced from the following command line invocation:

```
$ uFldShoreBroker --example or -e
```

Listing 20.3: Example configuration of the `uFldShoreBroker` application.

```
1 =====
2 uFldShoreBroker Example MOOS Configuration
3 =====
4
5 ProcessConfig = uFldShoreBroker
6 {
7     AppTick    = 4
8     CommsTick = 4
9
10    bridge = src=DEPLOY_ALL, alias=DEPLOY
11    bridge = src=DEPLOY_$V,   alias=DEPLOY
12
13    qbridge = RETURN
14
15    bridge  = src=UP_LOITER_$N, alias=UP_LOITER
16
17    // NOTE: The following line is shorthand for the next two.
18    // qbridge = FOOBAR
19    // bridge  = src=FOOBAR_ALL, alias=FOOBAR
20    // bridge  = src=FOOBAR_$V,   alias=FOOBAR
21
22 }
```

20.5 Publications and Subscriptions for uFldShoreBroker

The interface for `uFldShoreBroker`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldShoreBroker --interface or -i
```

20.5.1 Variables Published by uFldShoreBroker

The primary output of `uFldShoreBroker` to the `MOOSDB` are the requests to `pShare` for registrations, and the outgoing acknowledgement replies to remote node/vehicle communities.

- `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 20.3.
- `PMB_REGISTER`: A message to `pShare` to add a new bridge for a given variable and given target MOOS community at a specified IP address and port number.
- `NODE_BROKER_ACK`: A message written locally but bridged to a remote vehicle MOOS community, containing IP address and port information about the local shoreside community.

20.5.2 MOOS Variables Subscribed for by uFldShoreBroker

The `uFldShoreBroker` application subscribes to the following MOOS variables:

- `APPCAST_REQ`: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 11.10.4.
- `PHI_HOST_INFO`: Information about the local host IP address, the MOOS community name, the port on which the DB is running, and the port on which the local `pShare` is listening for UDP messages.
- `NODE_BROKER_PING`: Information published presumably by `uFldNodeBroker` running in a remote vehicle community. Message has information about the node host including the community name, IP address, the port number for the `MOOSDB`, input route(s) for the local `pShare` process.

20.5.3 Command Line Usage of uFldShoreBroker

The `uFldShoreBroker` application is typically launched with pAntler, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing:

```
$ uFldShoreBroker --help or -h
```

Listing 20.4: Command line usage for the `uFldShoreBroker` tool.

```
1 =====
2 Usage: uFldShoreBroker file.moos [OPTIONS]
```

```
3 =====
4
5 Options:
6   --alias=<ProcessName>
7       Launch uFldShoreBroker with the given
8       process name rather than uFldShoreBroker.
9   --example, -e
10      Display example MOOS configuration block.
11   --help, -h
12      Display this help message.
13   --interface, -i
14      Display MOOS publications and subscriptions.
15   --version,-v
16      Display release version of uFldShoreBroker.
```

21 uFldNodeComms: Simulating Intervehicle Communications



The `uFldNodeComms` application is a tool for handling node reports and messages between vehicles. Rather than directly sending node reports and messages between vehicles, `uFldNodeComms` acts as an intermediary to conditionally pass a report or message on to another vehicle, where conditions may be the inter-vehicle range or other criteria. The assumption is that `uFldNodeComms` is running on a topside or shoreside computer, and receiving information about the present physical location of deployed vehicles through node reports. The typical layout is shown in Figure 78.

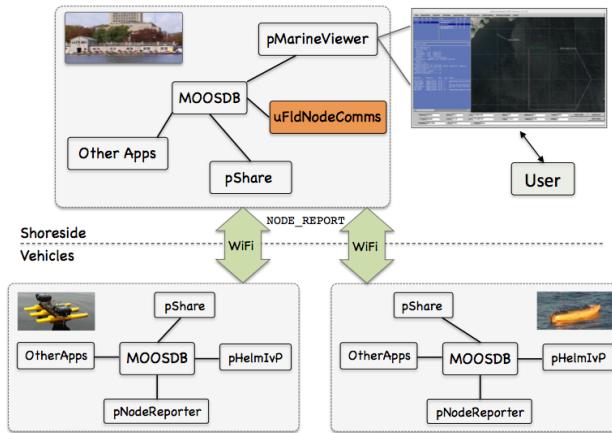


Figure 78: **Typical uFldNodeComms Topology:** A shoreside or topside community is receiving information from several deployed vehicles, in the form of node reports. The node reports contain time-stamped updated vehicle positions, from which the speed and distance measurements are derived and posted to the shoreside MOOSDB.



In short, `uFldNodeComms` subscribes for incoming node reports for any number of vehicles, and keeps the latest node report for each vehicle. On each iteration, for each vehicle, if the node report has been updated, the report is published to a specially created MOOS variable for the other n-1 vehicles. A user-configured criteria is applied before publishing the new information. Typically this criteria involves the range between vehicles, but the criteria may be further involved. The idea between three vehicles *alpha*, *bravo*, and *charlie* is shown below in Figure 79.

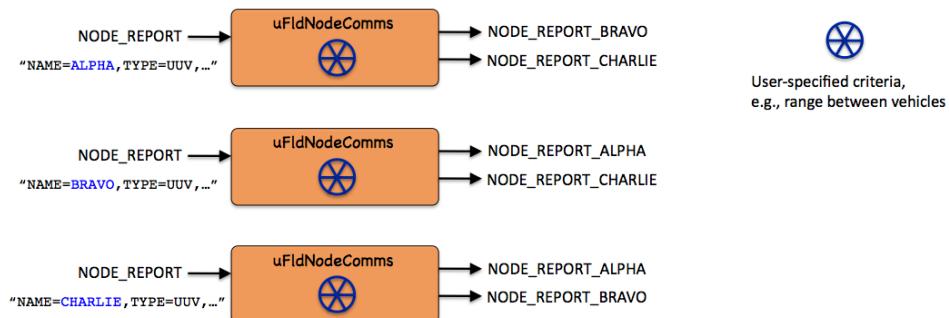


Figure 79: **Brokering by uFldNodeComms:** Each incoming node report is sent out to a specially named variable corresponding to one of the other n-1 vehicles.

21.1 Handling Node Reports

 Node reports may contain a bundle of useful information for sharing between vehicles. Perhaps the most important is the present vehicle position and trajectory. This may be used by the receiving vehicle for collision avoidance and formation keeping and so on. The vehicle position is also used by `uFldNodeComms` to determine if the node reports themselves are to be shared between vehicles. The inter-vehicle ranges derived from the `NODE_REPORT` messages are also used to determine if other more generic information contained in the `NODE_MESSAGE` variable is to be shared between vehicles.

21.1.1 The Criteria for Routing Node Reports

The basic criteria for sharing node reports is range. By default, the node report received for any one vehicle is passed on to *all* other known vehicles within comms range of the originating vehicle. The comms range is set by the parameter `comms_range` as on line 9 in Listing 3. Starting with this as a baseline, a few other factors may be involved in determining whether a node report is shared.

Using Vehicle Group Information

If `uFldNodeComms` is configured with the parameter `groups` set to true, as on line 21 in Listing 3, then node reports are only shared between vehicles having the same group name. The group name is a field contained in the node report itself, so the onus is on the vehicle to include this information as part of its report. The `pNodeReporter` application contains an optional configuration parameter `group=<group-name>` where the group information is declared for inclusion in all node reports. The motivation for the grouping option is to support multi-vehicle competitions where some vehicles want to convey positions to teammates, but not adversarial vehicles.

The `groups` feature only affects the passing of node reports between vehicles. It does not affect the passing of node *messages* discussed further below. Node messages contain their addressee information explicitly.

Establishing and Inter-Vehicle Critical Range

When the two vehicles are within a range deemed critical, as set by the `critical_range` configuration parameter as on line 10 in Listing 3, node reports are shared between vehicles regardless of the `comms_range` parameter and the `groups` parameter. The default for this parameter is 30 meters. The thought behind this feature is that, while it may be advantageous to not broadcast your own vehicle position to non group members for the purposes of a competition, it may be a good idea to share this information for the sake of collision avoidance.

Checking for Staleness

Since up-to-date inter-vehicle range information is used as part of the criteria in determining whether a vehicle receives a new node report from another, the position of the candidate recipient vehicle needs to reasonably up-to-date. The `stale_time` configuration parameter may be set as on line 11 in Listing 3, to determine the amount of elapsed time without receiving a node report from a vehicle before it is considered stale. If a recipient vehicle becomes stale, it will also not receive node messages.

Bestowing a Vehicle with an Enhanced Stealth Property

By default, node reports are shared at equal distances between sender and receiver. In other words, when one vehicle comes into close enough range to receive node reports of another, the other vehicle will also begin receiving node reports of the first vehicle. This puts each vehicle on equal footing if they are regarded in an adversarial context.

By using the

stealth parameter, one vehicle may be given a bit of an advantage. The stealth parameter is assigned to a particular vehicle and is a number in the range [0.5, 1]. A message from source to destination is sent if:

$$\text{actual_range}(\text{src}, \text{dest}) < \text{comms_range} * \text{stealth}(\text{src}) \quad (5)$$

By default, the stealth factor for each vehicle is 1. A stealthy vehicle with a factor of one half, means the receiving vehicle needs to be twice as close as it would otherwise to receive the stealthy vehicle's node report or node message. The `critical_range` parameter may be used to override a vehicle's stealthiness with respect to sending node reports. That is, a message from source to destination is sent if:

$$\text{actual_range}(\text{src}, \text{dest}) < \max((\text{comms_range} * \text{stealth}(\text{src})), \text{critical_range}) \quad (6)$$

The `stealth` value for a particular vehicle may be set in the MOOS configuration file as shown on line 18 in Listing 3. It may also be set dynamically by receiving mail on the variable `UNC_STEALTH`. For example the posting

```
UNC_STEALTH = vname=alpha, stealth=0.75
```

would immediately reset the stealth factor for vehicle *alpha* on the next iteration. The motivation for exposing this parameter via incoming MOOS mail is that another shoreside application, monitoring vehicle speed for example, may reward a vehicle operating in the field with increased stealth based on any criteria. This can be useful in constructing vehicle competitions.

Bestowing a Vehicle with an Enhanced Listening Property

In addition to the stealth property, a complementary property may be bestowed upon a vehicle using the `earange` parameter. The earange parameter is assigned to a particular vehicle and is a number in the range [1, 2]. A message from source to destination is sent if:

$$\text{actual_range}(\text{src}, \text{dest}) < \text{comms_range} * \text{earange}(\text{dest}) \quad (7)$$

By default, the earange factor for each vehicle is 1. A vehicle with a factor of two may receive node reports of another vehicle at twice the range it may otherwise. In the end the stealth and earange properties may cancel each other out when their extreme values are factored together. Taken together a message from source to destination is sent if:

$$\text{actual_range}(\text{src}, \text{dest}) < \text{comms_range} * \text{stealth}(\text{src}) * \text{earange}(\text{dest}) \quad (8)$$

The `earange` value for a particular vehicle may be set in the MOOS configuration file as shown on line 19 in Listing 3. It may also be set dynamically by receiving mail on the variable `UNC_EARANGE`. For example the posting

```
UNC_EARANGE = vname=alpha, earange=1.5
```

would immediately reset the earange factor for vehicle *alpha* on the next iteration. The motivation for exposing this parameter via incoming MOOS mail is that another shoreside application, monitoring vehicle speed for example, may reward a vehicle operating in the field with increased earange based on any criteria. This can be useful in constructing vehicle competitions.

21.1.2 Node Report Transmissions and pShare

Node reports are communicated to recipient vehicles in coordination with the `pShare` application. For each unique vehicle name discovered by `uF1dNodeComms` via received node reports, it will publish a new MOOS variable `NODE_REPORT_NAME`. This variable will be published with the contents of other vehicles' node reports.

As shown in Figure 79, if three vehicles, *alpha*, *bravo*, and *charlie* become known, three corresponding MOOS variables `NODE_REPORT_ALPHA`, `NODE_REPORT BRAVO`, and `NODE_REPORT CHARLIE` will be published. The variable `NODE_REPORT_ALPHA` will be published with reports from *bravo* and *charlie* and so on. To make this happen, `uF1dNodeComms` needs the corresponding share relationships from, for example, `NODE_REPORT_ALPHA` in the shoreside community to the variable `NODE_REPORT` in the alpha community on the alpha vehicle. This sharing relationship is established separately by the `uF1dShoreBroker` application running in the shoreside community.

21.2 Handling Node Messages

Node messages are of a generic structure for sharing a MOOS variable-value pair between a source node and a destination node. A node message from vehicle *alpha* to vehicle *bravo* would be posted locally by *alpha* with something like:

```
NODE_MESSAGE = "src_node=alpha,dest_node=bravo,var_name=FOOBAR,string_val=hello"
```

The local `NODE_MESSAGE` posting is shared to the shoreside community running `uF1dNodeComms` where it is considered for re-routing to the destination vehicle.

21.2.1 The Criteria for Routing Node Messages

The basic criteria for sharing node messages is (a) the message addressee, and (b) the range between the source and destination vehicles. Since `uF1dNodeComms` is receiving and keeping track of incoming node reports from all vehicles, it has ready access to the inter-vehicle range between the source and destination nodes. The same criteria used for sending node *reports* is used for sending node messages. It must meet the range criteria as perhaps modified by the stealth and earange factors discussed earlier. The only difference is that the `critical_range` parameter is irrelevant for the issue of sending node messages. This parameter was only used for node reports in the interest of safety and collision avoidance. There are however two other factors that may affect node message transmission, discussed next, message frequency and message size.

21.2.2 Enforcing a Minimum Time Between Node Messages

A maximum send frequency is enforced by requiring a minimum wait time between successful sends from a given source node. This minimum time is given by the parameter `min_msg_interval` as on line 13 in Listing 3. The default is 30 seconds. This interval is defined by the time starting with a successful transmission of a node message from a source to any destination. A separate log is kept by `uFldNodeComms` for each known vehicle.

21.2.3 Enforcing a Maximum Node Message Length

The length of node messages may be limited with the parameter `max_msg_length`, as on line 14 in Listing 3. The default maximum length is 1000 characters. The length of a message refers to the number of characters in the `string_val` field. For example, the length of the message

```
NODE_MESSAGE = "src_node=alpha,dest_node;bravo,var_name=FOOBAR,string_val=hello"
```

is five. Limiting the message length is a proxy for intervehicle communications where message packet length is constrained, as with acoustic communications for example.

21.2.4 Posting Messages to a Vehicle Group

A node message is typically addressed to another named vehicle. The sender may also address the message by group name. All vehicles in the group meeting the prevailing range criteria will receive the message. The group associated with the vehicle is declared in the node report sent by that vehicle. A node message using a group address is similar except for the use of the `dest_group` parameter

```
NODE_MESSAGE = "src_node=alpha,dest_group=red_team,var_name=FOOBAR,string_val=hello"
```

The sender has the option of indicating *both* a group name and vehicle name as the message destination. It may also specify more than one vehicle name explicitly. Thus the following is allowed:

```
NODE_MESSAGE = "src_node=alpha,dest_name;bravo:charlie:gilda,dest_group=red_team,
var_name=FOOBAR,string_val=hello"
```

If a destination vehicle is specified twice in the list of destinations or implicitly in the named group, the message will be sent only once to the destination vehicle.

21.3 Visual Artifacts for Rendering Inter-Vehicle Communications

Each time a node report or message is sent to a vehicle by `uFldNodeComms`, another posting is made to the variable `VIEW_COMM_PULSE`. This message may be subscribed for by another application using it to visually render the communications events. The screen shot in Figure 80 below shows four vehicles. The two bottom vehicles are sharing node reports indicated by the red and blue comms pulses.

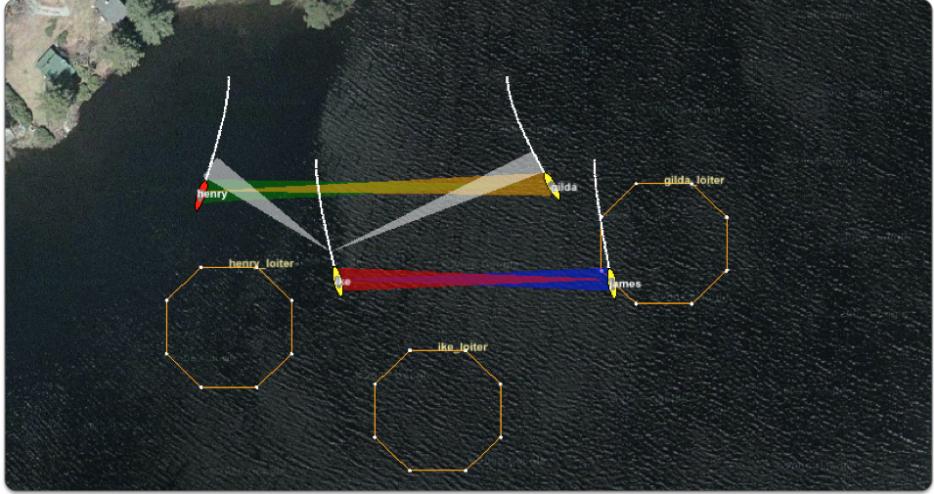


Figure 80: **Communication Pulses:** A visual artifact, a `VIEW_COMM_PULSE`, is rendered between vehicles when either a node report or node message is generated. The white pulse indicates a node message, and the non-white pulses indicate a node report. The pulse widens from a point at the source vehicle to maximum width at the destination vehicle. A pulse will remain rendered in pMarineViewer for some number of seconds after initial post, unless it is replaced by a new pulse with the same label.

The top two vehicles are also sharing node reports seen by the yellow and green pulses. The bottom left vehicle has also just sent a node message to the top two vehicles indicated by the two white pulse. Note the pair of white pulses were posted a few seconds prior to the present point in time since they point to vehicle positions just back in the vehicle history. The node reports are updated continuously, constantly replacing the pulse just previously posted.

The pulse colors are chosen automatically by `uFlreeNodeComms`. They may be toggled off in `pMarineViewer` with the '`o`' key. The comms pulse is conveyed in a posting to the variable `VIEW_COMM_PULSE`. The format is shown with the following example.

```
VIEW_COMM_PULSE = "sx=109.63,sy=-60.06,tx=30.96,ty=-60.22,beam_width=7,duration=10,
                    fill=0.35,label=JAMES2IKE,edge_color=green,fill_color=red,
                    time=2652414456.59,edge_size=1"
```

Comms pulses may be generated by other applications besides `uFlreeNodeComms`, and may be consumed and rendered by other applications besides `pMarineViewer`. The definition of the comms pulse object and the methods for serializing and de-serializing between object and string representation may be found in the `lib_geometry` library in the moos-ivp software tree.

21.4 Terminal and AppCast Output

The `uFlreeNodeComms` application produces some useful information to the terminal and identical content through appcasting. An example is shown in Listing 1 below. On line 2, the name of the local community (usually shoreside) is listed on the left. On the right, "0/0(339)" indicates there are no configuration or run warnings, and the current iteration of `uFlreeNodeComms` is 339. Lines 4-12 convey a summary of received and sent node reports from each vehicle. The number in parentheses at the end of lines 7 and 8 indicate the elapsed time since the last node report was received.

Listing 21.1: Example uFldNodeComms console and appcast output.

```
1 =====
2 uFldNodeComms shoreside                               0/0(339)
3 =====
4 Node Report Summary
5 =====
6     Total Received: 3101
7         GILDA: 1552      (0.0)
8         HENRY: 1549      (0.0)
9     -----
10    Total Sent: 628
11        GILDA: 315
12        HENRY: 313
13
14 Node Message Summary
15 =====
16    Total Msgs Received: 4
17        HENRY: 4      (24.1)
18     -----
19    Total Sent: 4
20        GILDA: 4
21     -----
22    Total Blocked Msgs: 0
23        Invalid: 0
24        Stale Receiver: 0
25        Too Recent: 0
26        Msg Too Long: 0
27        Range Too Far: 0
28
29 =====
30 Most Recent Events (4):
31 =====
32 [146.22]: Msg rec'd: src_node=henry,dest_node=gilda,var_name=UPDATE_LOITER,string_al=speed=2.4
33 [116.10]: Msg rec'd: src_node=henry,dest_node=gilda,var_name=UPDATE_LOITER,string_al=speed=2.4
34 [86.46]: Msg rec'd: src_node=henry,dest_node=gilda,var_name=UPDATE_LOITER,string_vl=speed=2.4
35 [56.32]: Msg rec'd: src_node=henry,dest_node=gilda,var_name=UPDATE_LOITER,string_val=speed=0.4
```

The summary of node messages is shown in the second half of the report, in lines 14-33 in this case. The total messages received is shown on line 16, with a breakdown of where they have been received from in the following lines. Starting on line 19, a summary of sent node messages is given. First the total sent messages on line 19 and a breakdown of receivers in the following lines. A summary of blocked messages is given next, in this case in lines 22-27. The total number of blocked messages is given first, followed by the possible reasons for blocking in lines 23-27. Finally, the most recent messages are shown as events in the last lines of the report.

21.5 Configuration Parameters of uFldNodeComms

The following parameters are defined for [uFldNodeComms](#).

Listing 21.2: Configuration Parameters for [uFldNodeComms](#).

- `comms_range`: Max range outside which inter-vehicle node reports and node messages will not be sent. Legal values: any numerical value. The default is 100 meters. Section [21.1.1](#).
- `critical_range`: Range in meters within which inter-vehicle node reports will be shared even if group membership would otherwise disallow. Legal values: any numerical value. The default is 30 meters. Section [21.1.1](#).
- `debug`: If true, further debugging information is produced to the terminal output. Legal values: true, false. The default is false.
- `earange`: A parameter in the range of [1,10] for extending the range a vehicle may otherwise hear node reports from other vehicles. The default is 1. Section [21.1.1](#).
- `groups`: If true, inter-vehicle node reports are shared only if two vehicles are in the same group. May be overridden if the two vehicles are within the critical range. The default is false. Section [21.1.1](#).
- `min_msg_interval`: The number of seconds required between message sends for any one source vehicle. The default is 30 seconds. Section [21.2.2](#).
- `max_msg_length`: The total number of characters that may be sent in a string component of a node message. The default is 1000. Section [21.2.3](#).
- `stealth`: A parameter in the range [0,1] for reducing the range other vehicles may otherwise hear the node reports from a source vehicle. The default is 0.1. Section [21.1.1](#).
- `stale_time`: Time in seconds after which a vehicle will not receive node reports or messages unless a node report has been received by that vehicle. The default is 5 seconds. Section [21.1.1](#).
- `verbose`: If true, status reports are displayed to the terminal during operation. The default is false.
- `view_node_rpt_pulses`: If true, comms pulses are rendered between vehicles whenever a node report successfully makes its way from one vehicle to another. The default is true. Section [21.3](#).

An Example MOOS Configuration Block

Listing 3 shows an example MOOS configuration block produced from the following command line invocation:

```
$ uFldNodeComms --example or -e
```

Listing 21.3: Example configuration of the `uFldNodeComms` application.

```
1 =====
```

```

2 uFldNodeComms Example MOOS Configuration
3 =====
4
5 ProcessConfig = uFldNodeComms
6 {
7     AppTick    = 4
8     CommsTick  = 4
9
10    comms_range      = 100          // default (in meters)
11    critical_range   = 30          // default (in meters)
12    stale_time       = 5           // default (in seconds)
13
14    min_msg_interval = 30          // default (in seconds)
15    max_msg_length   = 1000         // default (# of characters)
16
17    verbose   = true            // default
18
19    stealth   = vname=alpha, stealth=0.8
20    earange   = vname=alpha, earange=4.5
21
22    groups   = true
23
24    pulse_duration = 10;        // default (in seconds)
25 }

```

21.6 Publications and Subscriptions for uFldNodeComms

The interface for `uFldNodeComms`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldNodeComms --interface or -i
```

21.6.1 Variables Published by uFldNodeComms

The primary output of `uFldNodeComms` to the MOOSDB are the node reports and node messages out to the recipient vehicles, and visual artifacts to be used for rendering the inter-vehicle communications.

- **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 21.4.
- **NODE_MESSAGE_<VNAME>**: Node messages destined to be sent to a destination vehicle <VNAME> indicated as the recipient in the node message.
- **NODE_REPORT_<VNAME>**: Node reports destined to be sent to a given vehicle <VNAME> about the vehicle named in the node report.
- **VIEW_COMMs_PULSE**: A visual artifact for rendering the sending of a node report or node message between vehicles.

21.6.2 Variables Subscribed for by uFldNodeComms

The `uFldNodeComms` application subscribes to the following MOOS variables:

- **APPCAST_REQ**: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 11.10.4.
- **NODE_MESSAGE**: A node message from one vehicle to another.
- **NODE_REPORT**: A node report for a given vehicle from **pNodeReporter**.
- **NODE_REPORT_LOCAL**: Another name for a node report for a given vehicle from **pNodeReporter**.
- **UNC_STEALTH**: The extra stealth allowed a given vehicle to "hide" node reports to others.
- **UNC_EARANGE**: The extra range allowed a given vehicle to "hear" node reports of others.
- **UNC_VIEW_NODE_RPT_PULSES**: A way for external apps to tell **uFldNodeComms** to turn off or on the rendering of comms pulses whenever a node report is sent from one vehicle to another. Section 21.3.

Command Line Usage of uFldNodeComms

The **uFldNodeComms** application is typically launched with pAntler, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing:

```
$ uFldNodeComms --help or -h
```

*Listing 21.4: Command line usage for the **uFldNodeComms** tool.*

```

1 =====
2 Usage: uFldNodeComms file.moos [OPTIONS]
3 =====
4
5 Options:
6   --alias=<ProcessName>
7     Launch uFldNodeComms with the given process
8     name rather than uFldNodeComms.
9   --example, -e
10    Display example MOOS configuration block.
11   --help, -h
12    Display this help message.
13   --interface, -i
14    Display MOOS publications and subscriptions.
15   --version,-v
16    Display the release version of uFldNodeComms.
17
18 Note: If argv[2] does not otherwise match a known option,
19       then it will be interpreted as a run alias. This is
20       to support pAntler launching conventions.

```

22 uFldMessageHandler: Handling Incoming Node Messages

The `uFldMessageHandler` application is a tool for handling incoming inter-node messages. In the uField Toolbox typical arrangement, these messages are arriving from a shoreside MOOS community from the `uFldNodeComms` and `pShare` applications as shown below in Figure 81.

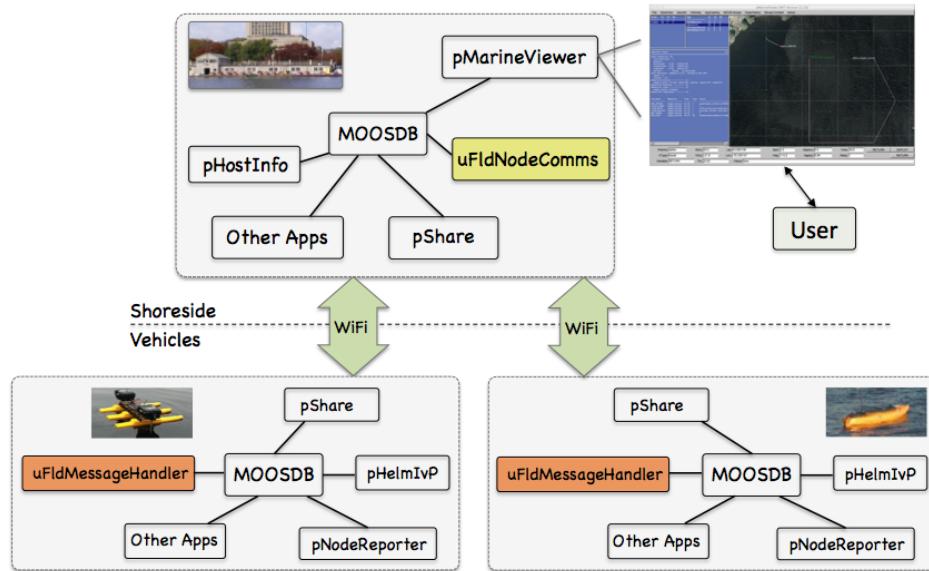


Figure 81: **Typical uFldMessageHandler Topology:** A vehicle (node) sends a message to another vehicle by wrapping the message content and addressee information in a single string sent to the shoreside. On the shoreside, the `uFldNodeComms` application redirects the message to the appropriate vehicle(s). The message is received on the vehicle by the `uFldMessageHandler` application which parses the MOOS variable and the variable value from the string and posts the variable-value pair to the local MOOSDB.

The functionality of `uFldMessageHandler` may be paraphrased:

- A source vehicle *alpha* wishes to send a message to vehicle *bravo* of the form `SPEED=2.5`.
- A local message is posted on vehicle *alpha* of the form:

```
NODE_MESSAGE_LOCAL = src_node=alpha,dest_node=bravo,src_var=SPEED,double_val=2.5
```

- The above message is bridged from *alpha* to the shoreside community using `pShare`.
- The message is received in the shoreside community as the variable `NODE_MESSAGE` and handled by `uFldNodeComms` and republished as `NODE_MESSAGE_BRAVO`.
- The message is then bridged out to *bravo* using `pShare` arriving in vehicle *bravo* as `NODE_MESSAGE`.
- On vehicle *bravo*, the `NODE_MESSAGE` is handled by `uFldMessageHandler`. The source variable and value are parsed and a post to the local MOOSDB on *bravo* is made, `SPEED=2.5`
- A scope on the MOOSDB on *bravo* would show the source of the `SPEED=2.5` posting to be "`uFldMessageHandler`", and the auxiliary source would show "alpha"

22.1 Configuration Parameters of uFldMessageHandler

The following parameters are defined for `uFldMessageHandler`.

Listing 22.1: Configuration parameters for `uFldMessageHandler`.

- `strict_addressing`: If true, only messages with a destination specified by `dest_node`, matching the local community name are processed. Other messages with a destination specified by a group designation are ignored. The default is false.
- `verbose`: If true, terminal output reports are generated on each iteration. The default is true.

An Example MOOS Configuration Block

Listing 2 shows an example MOOS configuration block produced from the following command line invocation:

```
$ uFldMessageHandler --example or -e
```

Listing 22.2: Example configuration of the `uFldMessageHandler` application.

```
1 =====
2 uFldMessageHandler Example MOOS Configuration
3 =====
4
5 ProcessConfig = uFldMessageHandler
6 {
7     AppTick    = 4
8     CommsTick = 4
9
10    strict_addressing = false // the default
11 }
```

22.2 Publications and Subscriptions for uFldMessageHandler

The interface for `uFldMessageHandler`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldMessageHandler --interface or -i
```

22.2.1 Variables Published by uFldMessageHandler

The primary output of `uFldMessageHandler` to the MOOSDB are the messages posted by parsing incoming `NODE_MESSAGE` postings. A summary is also posted periodically to recap message handling totals.

- **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 22.3.
- **UMH_SUMMARY_MSGS**: A summary of total messages, valid messages and rejected messages handled thus far.

22.2.2 Variables Subscribed for by uFldMessageHandler

The `uFldMessageHandler` application subscribes to the following MOOS variables:

- **APPCAST_REQ**: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 11.10.4.
- **NODE_MESSAGE**: Incoming node messages.

Command Line Usage of uFldMessageHandler

The `uFldMessageHandler` application is typically launched with pAntler, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing:

```
$ uFldMessageHandler --help or -h
```

Listing 22.3: Command line usage for the `uFldMessageHandler` tool.

```
1 =====
2 Usage: uFldMessageHandler file.moos [OPTIONS]
3 =====
4
5 SYNOPSIS:
6 -----
7   The uFldMessageHandler tool is used for handling incoming
8   messages from other nodes. The message is a string that
9   contains the source and destination of the message as well as
0   the MOOS variable and value. This app simply posts to the
11  local MOOSDB the variable-value pair contents of the message.
12
13 Options:
14   --alias=<ProcessName>
15     Launch uFldMessageHandler with the given process name
16     rather than uFldMessageHandler.
17   --example, -e
18     Display example MOOS configuration block.
19   --help, -h
20     Display this help message.
```

```

21 --interface, -i
22     Display MOOS publications and subscriptions.
23 --version,-v
24     Display the release version of uFldMessageHandler.
25
26 Note: If argv[2] does not otherwise match a known option,
27     then it will be interpreted as a run alias. This is
28     to support pAntler launching conventions.

```

22.3 Terminal and AppCast Output

The `uFldMessageHandler` application produces some useful information to the terminal and identical content through appcasting. An example is shown in Listing 4 below. On line 2, the name of the local community or vehicle name is listed on the left. On the right, "0/0(841)" indicates there are no configuration or run warnings, and the current iteration of `uFldMessageHandler` is 841. In lines 4-9, general tallies are shown of received, invalid, and rejected messages. In lines 11-15, the tallies for received messages sorted by source vehicle are shown. The variable-value columns reflect only the last received message.

Listing 22.4: Example appcast and terminal output of `uFldMessageHandler`.

```

1 =====
2 uFldMessageHandler gilda                               0/0(841)
3 =====
4 Overall Totals Summary
5 =====
6     Total Received Valid: 5
7             Invalid: 0
8             Rejected: 0
9     Time since last Msg: 101.3
10
11 Per Source Node Summary
12 =====
13 Source  Total   Elapsed  Variable  Value
14 -----  -----  -----  -----  -----
15 henry    5      101.3    RETURN    true
16
17 Last Few Messages: (oldest to newest)
18 =====
19 Valid Mgs:
19     src_node=henry,dest_node=gilda,var_name=UPDATE_LOITER,string_val=speed
20     src_node=henry,dest_node=gilda,var_name=UPDATE_LOITER,string_val=speed
21     src_node=henry,dest_node=gilda,var_name=UPDATE_LOITER,string_val=speed
22     src_node=henry,dest_node=gilda,var_name=UPDATE_LOITER,string_val=speed
23     src_node=henry,dest_node=gilda,var_name=RETURN,string_val=true
24 Invalid Mgs:
25     NONE
26 Rejected Mgs:
27     NONE

```

The information group starting on line 17 shows the last five received valid, invalid and rejected messages. Note that a rejected message may be rejected for being invalid, or if the destination field

doesn't match, or if strict addressing is enabled and there is not a precise destination field match.

23 uFldScope: Gathering a Multi-Vehicle Status Summary

The **uFldScope** application is a tool for collecting diverse sets of information regarding a field of vehicles remotely deployed. Suppose, for example, one is interested in monitoring, for each deployed vehicle, the (a) helm mode, (b) total distance travelled, (c) battery level, and (d) the number of times it has visited a certain beacon. Each piece of information may be embedded in one of a number of MOOS variables, perhaps along with a lot of other information of no concern. For example, a typical **NODE_REPORT** posting contains the helm mode, but the full string may look like:

```
NODE_REPORT= "NAME=alpha,TYPE=UUV,TIME=1252348077.59,X=51.71,Y=-35.50,LAT=43.824981,
    LON=-70.329755,SPD=2.00,HDG=118.85,DEPTH=4.63,LENGTH=3.8,MODE=LOITERING"
```

While there are several methods to scope on the above variable and pick out the helm mode, the goal of the **uFldScope** tool is to have this information readily visible for each vehicle perhaps alongside other key fields for all vehicles, in a continuously updated simple table like the following:

VName	MODE	TripDist	Speed	STREAMING(2)
=====	=====	=====	=====	(15)
alpha	LOITERING	66.8	1.96	
bravo	PARK	0.0	0.00	
charlie	RETURNING	1466.3	1.05	

The assumption is that **uFldScope** is running on a topside computer, interacting with a user, and receiving information on deployed vehicles primarily through node reports or other summary report variables. The typical layout is shown in Figure 82

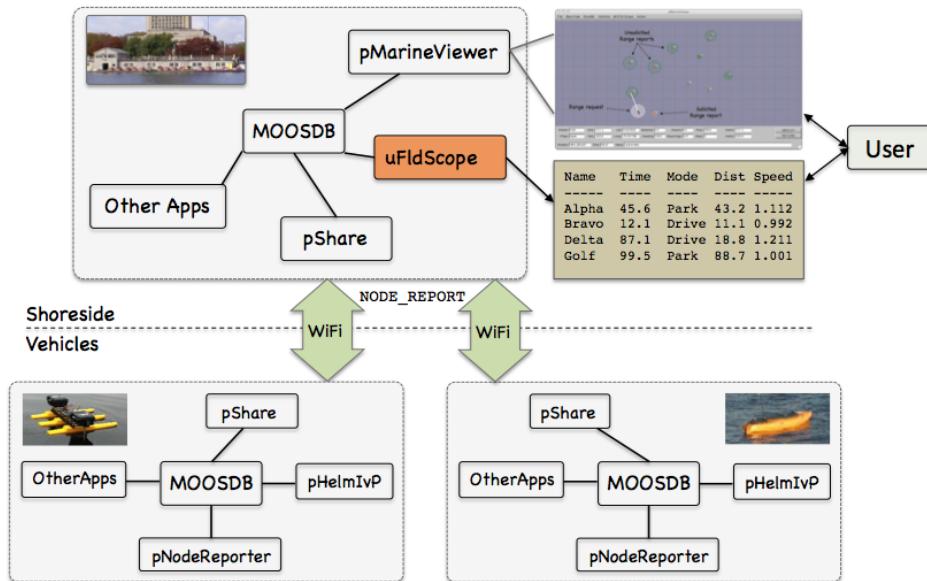


Figure 82: **Typical uFldScope Topology:** A shoreside or topside community is receiving information from several deployed vehicles. Key information is embedded in one of several possible MOOS report variables. The uFldScope tool runs in the topside community to parse key information from the variables and display them in a table format configured by the user.

23.1 Configuration Parameters of uFldScope

The following parameters are defined for `uFldScope`. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so.

Listing 23.1: Configuration parameters for `uFldScope`.

`scope`: Description info include in main report along with source info.

`layout`: An alternative table layout showing only selected fields in each report.

23.1.1 An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command-line:

```
$ uFldScope --example or -e
```

This will show the output shown in Listing 2 below.

Listing 23.2: Example configuration of the `uFldScope` application.

```
1 =====
2 uFldScope Example MOOS Configuration
3 =====
4
5 ProcessConfig = uFldScope
6 {
7     AppTick    = 4
8     CommsTick = 4
9
10    scope = var=NODE_REPORT,key=vname,fld=TIME,alias=Time
11    scope = var=NODE_REPORT,key=vname,fld=MODE
12    scope = var=SPEED_REPORT,key=vname,fld=avg_speed,alias=speed
13    scope = var=ODOMETRY_REPORT,key=vname,fld=trip_dist
14    scope = var=ODOMETRY_REPORT,key=vname,fld=total_dist
15
16    layout = trip_dist, total_dist
17    layout = MODE, speed, Time
18 }
```

23.2 Publications and Subscriptions for uFldScope

The interface for `uFldScope`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldScope --interface or -i
```

23.2.1 Variables Published by uFldScope

The primary output of `uFldScope` to the MOOSDB is the report output to the terminal or appast. The only publication to the MOOSDB is the `APPCAST` publication.

23.2.2 MOOS Variables Subscribed for by uFldScope

The `uFldScope` application subscribes to the following MOOS variables:

- `APPCAST_REQ`: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 11.10.4.

The `uFldScope` application will also subscribe for the MOOS variables prescribed in the `scope` configuration parameter(s).

Command Line Usage of uFldScope

Although the `uFldScope` application may be launched with `pAntler`, it is typically launched separately from the command line:

```
$ uFldScope --help or -h
```

This will show the output shown in Listing 3 below.

Listing 23.3: Command line usage for the `uFldScope` tool.

```
1 Usage: uFldScope file.moos [OPTIONS]
2
3 Options:
4   --alias=<ProcessName>
5     Launch uFldScope with the given process
6     name rather than uFldScope.
7   --example, -e
8     Display example MOOS configuration block
9   --help, -h
10    Display this help message.
11   --version,-v
12    Display the release version of uFldScope.
```

23.3 Configuring the uFldScope Utility

The `uFldScope` utility may be configured to choose (a) which MOOS variables are scoped, (b) which fields in those messages are scoped, and (c) how that data is presented to the user. In all usage scenarios it is presumed that the messages are strings comprised of comma-separated variable=value pairs. For example:

```
NODE_REPORT= "NAME=alpha,TYPE=UUV,TIME=1252348077.59,X=51.71,Y=-35.50,LAT=43.824981,
LON=-70.329755,SPD=2.00,HDG=118.85,DEPTH=4.63,LENGTH=3.8,MODE=LOITERING"
```

For each variable specified by the user for scoping, a *key* needs to also be specified identifying the vehicle name. In the above case, the key is the string "NAME".

23.3.1 Configuring Scope Elements

A *scope element* corresponds to particular MOOS message and a particular field in that MOOS message. It also corresponds to a column in the tabular report presented to the user. For example, the second column in the table below, with the header "MODE", corresponds to a scope element deriving its information from postings to [NODE_REPORT](#), in the field "MODE".

VName	MODE	TripDist	Speed	STREAMING(2)
=====	=====	=====	=====	(15)
alpha	LOITERING	66.8	1.96	
bravo	PARK	0.0	0.00	
charlie	RETURNING	1466.3	1.05	

A scope element is configured in the mission configuration file with entries of the form:

```
scope = var=<MOOSVar>, key=<KEYNAME>, fld=<FIELDNAME>, alias=<ALIAS>
```

The <MOOSVAR> specifies the name of the MOOS variable. Multiple scope elements may use the same MOOS variable. The <KEYNAME> specifies the field in the message used to designate the name of the vehicle. There should be only one right answer for this, and if it is wrongly specified, the column for that scope element will simply be empty. The <FIELDNAME> specifies the other field in the message holding the information of interest. For example, line 10 in Listing 2 is the scope configuration resulting in the second column of the above tabular output. The <ALIAS> is a string to use in the column output header if the user doesn't want to just use the name of scoped field.

23.3.2 Configuring Scope Layouts

By default the tabular output produced by [uFldScope](#) contains a column for each scope element. If the number of scope elements is large the user may be interested, at times, in rendering only a subset of the scope elements. The user may define these subsets using the LAYOUT configuration parameter, of the form:

```
LAYOUT = <FLDNAME>, <FLDNAME>, ..., <FLDNAME>
```

The <FLDNAME> specifies the name of the field in a given scope element. Note that it is *possible* that the field names may be the same for two different scope elements. For this reason the field name used in the layout definition is the field name alias. This gives the user the chance to distinguish two otherwise identical field names. If the user does not specify an alias in configuration of scope element, the alias is by default the same as the field name.

For the [uFldScope](#) configuration shown in Listing 2, the following five scope elements are rendered to the user as follows.

VName	Time	MODE	speed	trip_dist	total_dist	PAUSED(A)
=====	=====	=====	=====	=====	=====	(1800)
gilda	2651922526.73	MODE@ACTIVE:LOITERING	1.06	517.1	517.1	
henry	2651922526.54	MODE@ACTIVE:LOITERING	1.17	526.4	526.4	
ike	2651922526.45	MODE@ACTIVE:LOITERING	1.07	520.1	520.1	
james	2651922526.65	MODE@ACTIVE:LOITERING	1.16	515.1	515.1	

Note the "(A)" at the end of the first line. This indicates that all scope elements are being presented. If the user hits the '1' or 'L' keys the presentation will be toggled through the various layouts configured by the user. In this example case, the following two layouts may be selected:

VName	Time	MODE	speed	PAUSED(1)
gilda	2651923071.3	MODE@ACTIVE:LOITERING	1.17	(1802)
henry	2651923071.11	MODE@ACTIVE:LOITERING	1.17	
ike	2651923071.03	MODE@ACTIVE:LOITERING	1.18	
james	2651923071.24	MODE@ACTIVE:LOITERING	1.17	

Note that only the *Time*, *MODE*, and *speed* scope elements are produced, corresponding to the layout configured on Line 16 in Listing 2. Also note that the "(A)" on the first line switched to "(1)" to indicate that the first user-configured layout is being used. By hitting 'L' once more, the second user-configured layout will be instead shown:

VName	trip_dist	total_dist	PAUSED(2)
gilda	1081.9	1081.9	(1801)
henry	1072.1	1072.1	
ike	1117.7	1117.7	
james	1067.0	1067.0	

Note that the "(1)" on the first line switched to "(2)" to indicate that the second user-configured layout is being used. By hitting 'L' once more, the full table will again be rendered.

23.3.3 Further Control of the Terminal Output

If multiple layouts have been configured, the user may either toggle through the list of layouts with the '1' or 'L' key as mentioned above, or toggle between the last-used user-configured layout and the mode of rendering all scope elements, by using the 'a' or 'A' key.

By default the output produced to the terminal is refreshed on each iteration of the `uFldScope` application. This may be useful for watching a trend as time passes. The user may also wish to pause the output to take a careful look at the data. This may be done by hitting the 'p' or 'P' keys, or simply the spacebar. Subsequent similar keystrokes will keep the refresh mode in the paused mode, but the output will be refreshed to their current values before again pausing. Returning to the streaming mode may be done by hitting the 'r' or 'R' keys. At any time the user may also hit the 'h' or 'H' keys for a help menu.

24 uFldPathCheck: Monitoring Vehicle Path Properties

The `uFldPathCheck` application is a tool for summarizing a few properties in a field of vehicles remotely deployed. The primary focus is on summarizing the speed and distance travelled for each vehicle. Rather than relying on the vehicles themselves to calculate and report this information, the `uFldPathCheck` tool determines this information independently based on node reports from the vehicles. The assumption is that `uFldPathCheck` is running on a topside or shoreside computer, and receiving information about deployed vehicles primarily through node reports. The typical layout is shown in Figure 83

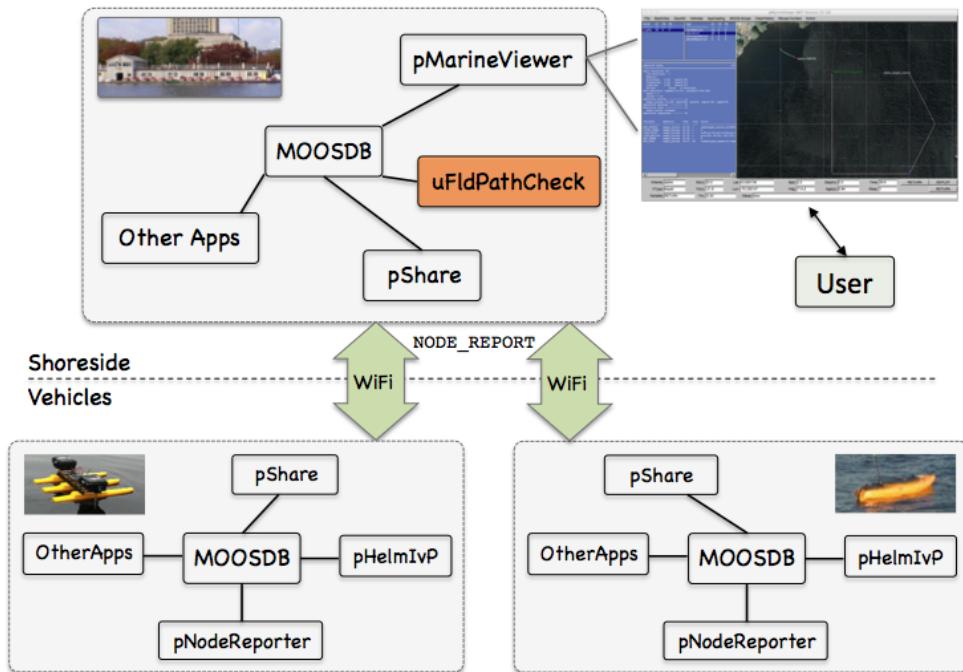


Figure 83: **Typical uFldPathCheck Topology:** A shoreside or topside community is receiving information from several deployed vehicles, in the form of node reports. The node reports contain time-stamped updated vehicle positions, from which the speed and distance measurements are derived and posted to the shoreside MOOSDB.

In short, `uFldPathCheck` subscribes for incoming node reports for any number of vehicles, and keeps a running history of positions for each vehicle. It uses this recent-history to post (a) the current speed noted per vehicle and (b) the distance travelled per vehicle. These two reports are posted in the `UPC_SPEED_REPORT` and `UPC_ODOMETRY_REPORT` variables to the MOOSDB. The odometry report includes a total distance travelled, and a "trip-ometer" distance travelled since the last trip reset. The trip-ometer may be reset for a vehicle *alpha* when mail is received of the form `UPC_TRIP_RESET=alpha`. Examples of the posted output form are given below in Section 24.2.1.

24.1 Overview of the uFldPathCheck Interface and Configuration Options

The `uFldPathCheck` application may be configured with a configuration block within a `.moos` file. Its interface is defined by its publications and subscriptions for MOOS variables consumed and

generated by other MOOS applications. An overview of the set of configuration options and interface is provided in this section.

24.1.1 Configuration Parameters of uFldPathCheck

The following parameters are defined for `uFldPathCheck`.

Listing 24.1: Configuration parameters for `uFldPathCheck`.

`history`: Length of queue used for calculating present speed. The default is 10.

24.2 Publications and Subscriptions for uFldPathCheck

The interface for `uFldPathCheck`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldPathCheck --interface or -i
```

24.2.1 Variables Published by uFldPathCheck

The primary output of `uFldPathCheck` to the MOOSDB are the speed and odometry reports.

- `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 ??
- `UPC_ODOMETRY_REPORT`: The current odometry information for a given vehicle.
- `UPC_SPEED_REPORT`: The current speed for a given vehicle.

An example of the odometry report:

```
UPC_ODOMETRY_REPORT = "vname=alpha,total_dist=4205.4,trip_dist=1105.2"
```

An example of the odometry report:

```
UPC_SPEED_REPORT = "vname=alpha,avg_speed=2.21"
```

24.2.2 Variables Subscribed for by uFldPathCheck

The `uFldPathCheck` application subscribes to the following MOOS variables:

- `APPCAST_REQ`: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 11.10.4.
- `NODE_REPORT`: A node report for a given vehicle from `pNodeReporter`.
- `NODE_REPORT_LOCAL`: A node report for a given vehicle from `pNodeReporter`.
- `UPC_TRIP_RESET`: The name of a vehicle to have its trip odometer reset.

24.2.3 An Example MOOS Configuration Block

An example MOOS configuration block can be obtained by entering the following from the command-line:

```
$ uFldPathCheck --example or -e
```

Listing 24.2: Example configuration for `uFldPathCheck`.

```
1 =====
2 uFldPathCheck Example MOOS Configuration
3 =====
4
5 ProcessConfig = uFldPathCheck
6 {
7     AppTick    = 4
8     CommsTick = 4
9
10    history   = 10 (Default)
11 }
```

24.3 Usage Scenarios the `uFldPathCheck` Utility

The motivation for this tool is to have a module running on the shoreside capable of being used in a competition scenario. Especially in a simulated competition, there may be a need to limit the upper speed of a participating vehicle to ensure a level playing field between competitors. Since the vehicle simulators may be running on separate machines with participants merely reporting node reports, there is no way to directly control the upper speed of the vehicles. By *monitoring* the upper speed, this leaves open the option of either (a) disqualifying a vehicle caught moving too fast, or (b) imposing a penalty on a speeding vehicle. The penalty chosen is outside the scope of this module but may include reducing the number of points awarded for certain accomplishments, adding more noise to simulated sensors, and so on. Since this usage scenario implies a "non-compliant" vehicle, we don't want to base the monitored speed on a value reported by the vehicle. Instead we calculate the speed based on successive time-stamped node reports with positions.

Likewise, the odometry information calculated and posted is also intended for use in a competition context. Conceivably, part of a competition may require a vehicle to periodically "re-fuel" after travelling a certain distance. By having the odometry information calculated independently on the shoreside, constraints on total distance, or fuel calculations may be generated to impose on vehicle competitions. The `uFldPathCheck` application accepts the `UPC_ODOMETRY_RESET=VNAME` posting to reset the trip-ometer for a given vehicle, presumably after a re-fueling event is noted. The odometry information may also be used directly in competitions where minimizing the total path length is the primary objective.



25 uFldHazardSensor: Simulating an Simple Hazard Sensor

The `uFldHazardSensor` application is a tool for simulating an on-board sensor that processes sonar image data and (a) detects image components that may represent a hazard, and (b) further classifies the detected components as being either a hazard or benign object. The idea is shown in Figure 84. The user configures the sensor by choosing one of N swath widths available to the given sensor, and by choosing a probability of detection, P_D . Based on these two user choices, and the particular performance characteristics of the sensor and sensor algorithms, a probability of false alarm, P_{FA} and probability of correct classification, P_C , follow.

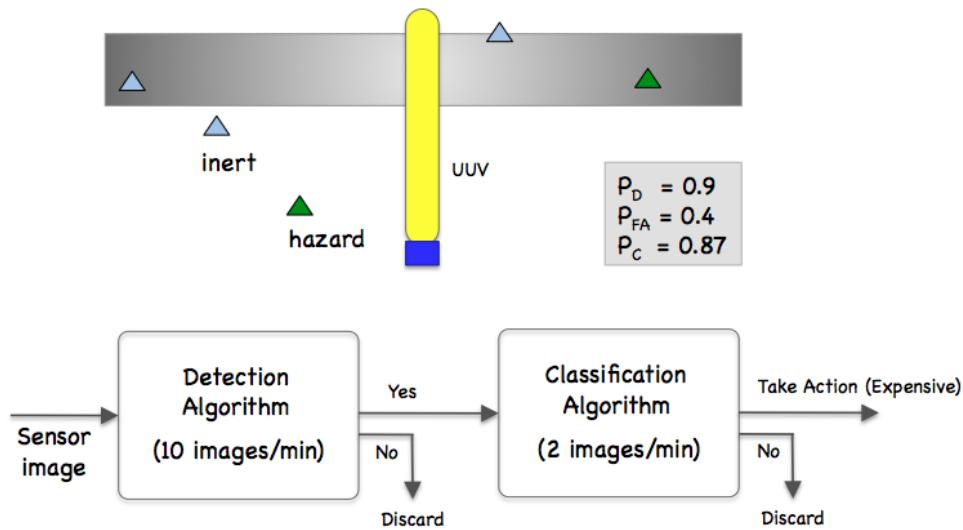


Figure 84: **Simulated Hazard Sensor:** A vehicle processes a series of sensor images which may or may not contain an object. The detection algorithm processes the image and rejects images it believes does not contain hazardous objects. It passes on images containing possible hazardous objects to a classifier, which makes a determination for each object in an incoming image as to whether or not the object is hazardous or benign.

In the `uFldHazardSensor` application, the objects and their properties (position, type, etc.) are read from a pre-generated *hazard file*.

The vehicle locations are known to the simulator from *node reports* received from the vehicles. And sensor output is sent to the vehicle in tidy `UHZ_DETECTION_REPORT` and `UHZ_HAZARD_REPORT` messages as a proxy to the actual hazard sensor and the calculations that would otherwise reside on the vehicle. The simulated sensor is configured to have a finite list of sensor settings, each consisting of a swath width, ROC curve, and probability of correct classification. It's up to the user to choose the sensor setting and choose a P_D on the associated ROC curve, which determines the prevailing P_{FA} .

25.1 A Quick Start Guide to Using uFldHazardSensor

To get started, we (a) point you to an example mission using `uFldHazardSensor`, (b) lay out the absolute minimum `uFldHazardSensor` configuration components, i.e., those which do not have default values, and (c) discuss the structure of a simple hazard file representing the ground-truth set of objects used by the simulated sensor.

25.1.1 A Working Example Mission - the Jake Mission

The example mission is referred to as the Jake example mission and may be found and launched in the moos-ivp distribution with:

```
$ cd moos-ivp/ivp/missions/m10_jake
$ ./launch.sh 12
```

This launches the simulation with time warp 12. The time warp may be adjusted to suit your preference and is bounded above by your computer's processing capability. After launching and hitting the deploy button, you should see something similar to Figure 85.

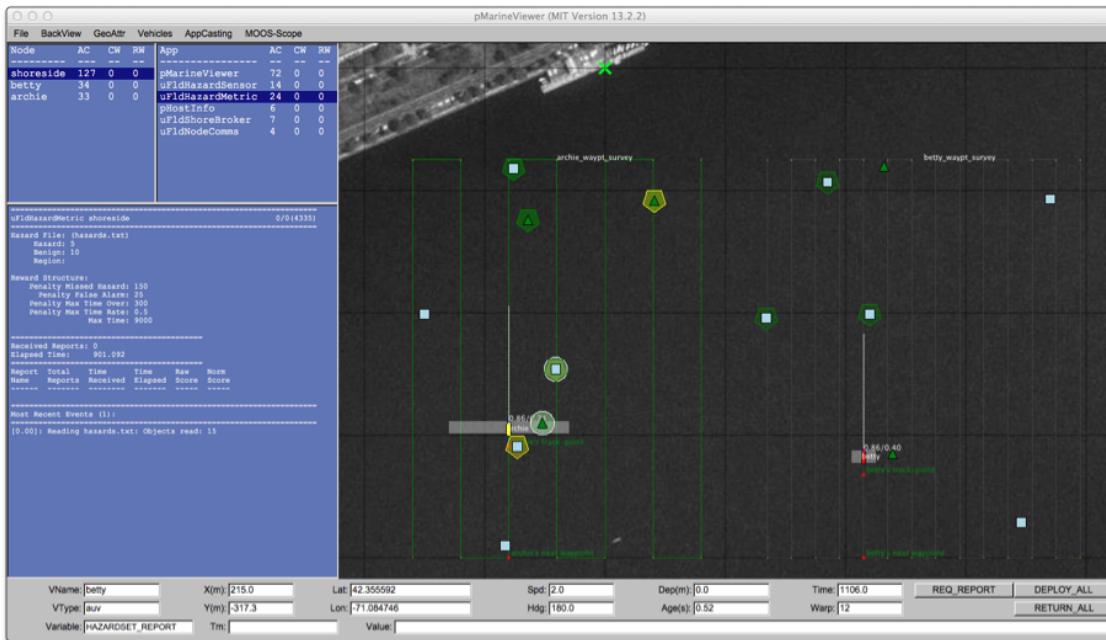


Figure 85: The Jake Example Mission: The jake example mission is involves a two vehicles each using the hazard sensor, traversing a search area using a lawnmower pattern. Both vehicles process their information and generate a hazardset report when finished.

There are few notable components of this similation that comprise the hazard search nature of the mission:

- **uFldHazardSensor:** The simulated sensor itself.
- **uFldHazardMgr:** A MOOS app running on the shoreside that will grade the incoming hazardset reports generated by the vehicle. This is described in detail in Section 26.
- **hazards.txt:** A text file representing the ground truth laydown of objects in the search area. (Section 25.1.3 and 25.4).
- **uFldHazardMetric:** A strawman MOOS app running on each vehicle to request sensor information, process the information, and generate a hazardset report upon request or the end of

the mission. (Section 27).

25.1.2 A Bare-Bones Example `uFldHazardSensor` Configuration

Listing 1 below shows a bare-bones configuration. Line 3 names a hazard file, containing the ground truth description of objects in the field. This format is described in Section 25.4, and an example hazard file is in the same directory as the Jake example mission.

Listing 25.1: Example bare-bones configuration of `uFldHazardSensor`.

```
1 ProcessConfig = uFldHazardSensor
2 {
3     hazard_file    = hazards.txt
4
5     sensor_config = width=25, exp=4, pclass=0.80
6     sensor_config = width=50, exp=2, pclass=0.60
7     sensor_config = width=10, exp=6, pclass=0.93
8 }
```

The possible sensor configuration options are listed in lines 5-7. These values are from the Jake example mission and perhaps are reasonable values for any mission, but they are not the default. These options, a `hazard_file` and at least one `sensor_config` option, *must* be specified or the sensor will not operate. The `sensor_config` parameter is discussed in Section 25.3.2. The full set of configuration parameters for `uFldHazardSensor` is provided in Section 25.9, along with an extended example configuration file.

25.1.3 A Simple Hazard File

In the Jake example mission, search is performed over a region containing a number of objects, some hazards, and some benign. The type and location of objects is defined in a *hazard file*. The hazard file for the Jake mission is shown below, and contains a single line for each object. The first line in the file is a copy of the command-line invocation used in generating this file. This command-line tool, `gen_hazards` is discussed in Section 25.4.2. An object in the file is comprised minimally of four components:

- The `x` location of the object,
- The `y` location of the object,
- The `label` of the object, unique across all objects,
- The `type` of the object.

```
// gen_hazards --polygon=-150,-75:-150,-400:400,-400:400,-75 --objects=5,hazard --objects=10,benign
hazard = x=-52,y=-290,label=75,type=hazard
hazard = x=41,y=-108,label=53,type=hazard
hazard = x=-64,y=-124,label=98,type=hazard
hazard = x=232,y=-80,label=91,type=hazard
hazard = x=239,y=-315,label=19,type=hazard
```

```

hazard = x=-41,y=-246,label=82,type=benign
hazard = x=185,y=-93,label=64,type=benign
hazard = x=-150,y=-201,label=11,type=benign
hazard = x=-76,y=-82,label=14,type=benign
hazard = x=-73,y=-309,label=28,type=benign
hazard = x=346,y=-371,label=73,type=benign
hazard = x=-83,y=-390,label=77,type=benign
hazard = x=220,y=-201,label=15,type=benign
hazard = x=134,y=-204,label=95,type=benign
hazard = x=370,y=-107,label=99,type=benign

```

Each object may contain additional components discussed in later sections.

25.1.4 Typical Simulator Topology

The typical module topology is shown in Figure 86 below. Multiple vehicles may be deployed in the field, each periodically communicating with a shoreside MOOS community running a single instance of `uFldHazardSensor`. Each vehicle regularly sends a node report to the shoreside community read by `uFldHazardSensor`. The hazard sensor also is launched with a file indicating the actual simulated hazard field, with each object having a location, classification and label. In short the hazard sensor knows ground truth, the location and orientation of all vehicles and the sensor settings for all vehicles.

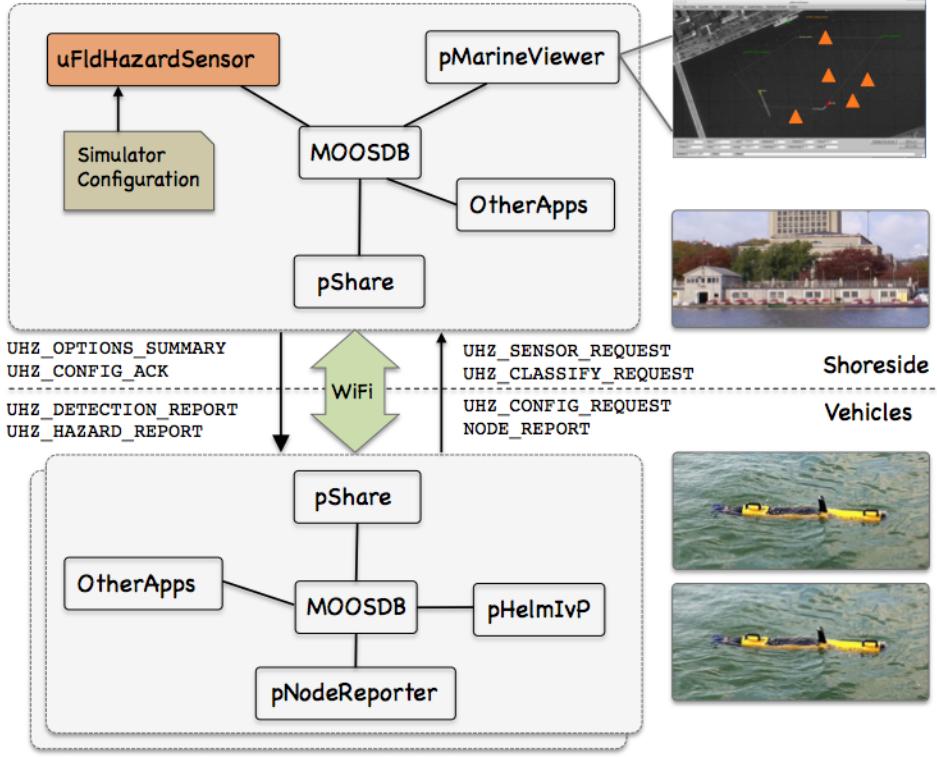


Figure 86: **Typical uFldHazardSensor Topology:** The simulator runs in a shoreside computer MOOS community and is configured with a hazard field containing both hazardous and benign objects. Vehicles accessing the simulator send a steady stream of messages (`UHZ_SENSOR_REQUEST`) and node reports to the shoreside community. The simulator continuously checks the connected vehicle's position against objects in the hazard field, and the sensor settings. When/if an object comes into sensor range, the simulator rolls the dice and if a detection is made, will send a `UHZ_DETECTION_REPORT` message to the vehicle. The vehicle may periodically re-configure its sensor setting by posting to `UHZ_CONFIG_REQUEST`. If the configuration request is acceptable, the simulator will respond with a message to `UHZ_CONFIG_ACK` bridged back out to the vehicle.

A vehicle operates its sensor by sending a steady stream of messages to the sensor on the shoreside under the variable `UHZ_SENSOR_REQUEST`. Each time the simulator receives this request, it will assess the vehicle's current position, and sensor settings, and may (depending on how the dice are rolled and if a detection is made) post a `UHZ_DETECTION_REPORT` to be bridged back out to the given vehicle. If the sensor does not make a detection, no report is sent.

Once the vehicle has made a detection, it may then make a further request of the hazard sensor to apply the classification algorithm to the given detection, by posting to variable `UHZ_CLASSIFY_REQUEST`, which also is bridged to the shoreside and handled by the hazard sensor. Regardless of the outcome, the hazard sensor will post a report, `UHZ_HAZARD_REPORT` indicating whether the detected object was either a hazard or a benign object. The frequency in which classification requests are handled is limited, by default to once per 30 seconds, but may be changed by setting the `min_classify_interval` parameter in the `uFldHazardSensor` configuration.

If running a pure simulation (no deployed vehicles), both MOOS communities may simply be running on the same machine configured with distinct ports. The `pShare` application is shown

here for communication between MOOS communities, but there are other alternatives for inter-community communication and the operation of `uFldHazardSensor` is not dependent on the manner of inter-communication communications.

25.2 Detections and Classifications

Each time a vehicle passes over an object in the field, with the object entering the vehicle's sensor swath and exiting again, we refer to this as a *sensor event*, depicted in Figure 87. Although the simulated sensor does not deal with sensor images, the actions of the sensor simulator after each event may be thought as dealing with a hypothetical sensor image of the object gathered while it was in the sensor swath.

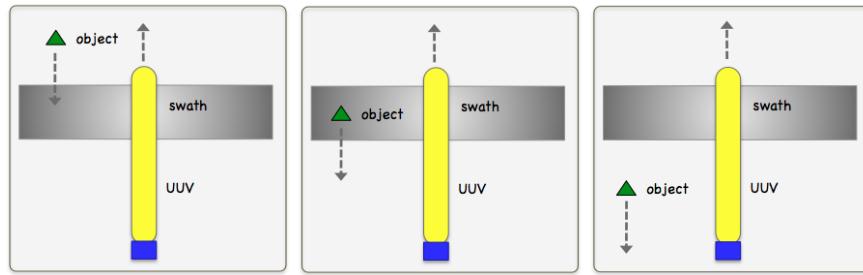


Figure 87: **A Sensor Event:** An sensor event occurs when an object enters the sensor swath. Following events for the same object may occur later after the object has exited the sensor swath.

For each sensor event, the simulator may generate two reports, a *detection report* and a *classification report*. The first is generated and returned to the vehicle unsolicited, while the latter is generated by the simulator only if the vehicle requests a classification report. The likelihood of a detection report and the likelihood of a correct classification are dependent on (a) the sensor settings, (b) the properties of the object being sensed, and (c) the relative angle of the vehicle sensor swath as it passes over the object.

25.2.1 The Simulated Detection Algorithm

The sensor simulator handles multiple vehicles and multiple vehicle sensor setting choices. For each vehicle, the simulator is receiving node reports (`NODE_REPORT`), allowing the simulator to know each vehicle's present position. Of course the simulator also knows the hazard field configuration. Armed with these three pieces of information, (a) the sensor setting, (b) the vehicle position, and (c) the hazard field, it repeatedly operates in the manner shown in Figure 88 below.

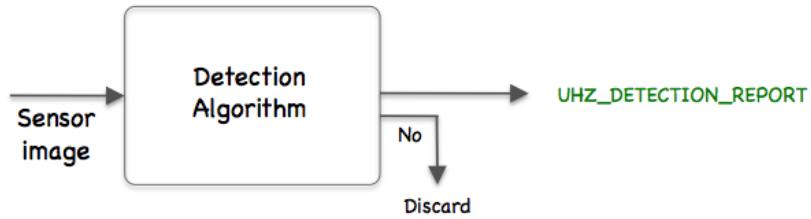


Figure 88: The simulated detection algorithm: Each time an object comes into the sensor field, it is considered once for detection. Once it goes out of the field, it may be processed again later if it comes back into the sensor field. If it passes the detection threshold, a detection report is posted to the MOOSDB. Otherwise no action is taken.

Each time an object comes into the sensor field for a vehicle, the simulator makes a detection determination based on:

- If the object is a hazard (ground-truth identified as such in the hazard configuration file), the simulator will report a detection with probability P_D .
- If the object is benign the simulator will report a detection with probability P_{FA} .

Again, the sensor swath width and P_D are chosen by the vehicle, and P_{FA} is set based on the chosen P_D and sensor characteristics. This is discussed in detail in Section ???. If no detection is made, no further communication or action is taken by the simulator. If a detection *is* declared, a report is generated by the simulator to be shared to the vehicle:

```
UHZ_DETECTION_REPORT_ARCHIE = x=51,y=11.3,label=12
```

Note this report contains the object's unique identifier which relieves the user of the problem of data association.

25.2.2 The Simulated Classification Algorithm

Once a detection has been made for an object, the object may be processed by the simulated classification algorithm. The idea is that the same set of raw sensor data, or image, is first passed through a quick detection filter, and the images for selected detections are then passed through a second, more resource-intensive, classification filter. This process may also be thought of as a proxy for offloading images through acoustic communications to another platform for processing. Either way, the idea is that the detection algorithm is fast, able to handle all raw data as it comes in, and the classification algorithm is more accurate but slower, forcing the careful consideration of which images to process.

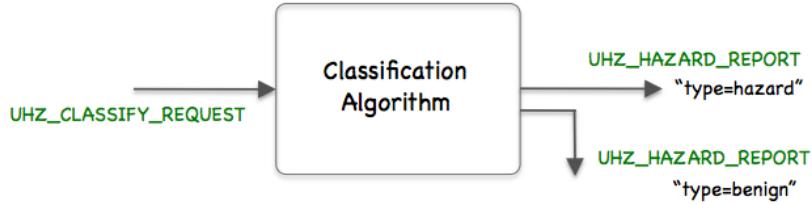


Figure 89: **Simulated classification algorithm:** The data from a given detection is processed by a separate algorithm for classifying the object as either a hazard or benign.

A classification request is made by the vehicle with a posting similar to the following:

```
UHZ_CLASSIFY_REQUEST = vname=archie,label=152
```

This variable would be published by the vehicle, named *archie*, and would be shared to the shoreside community. The requesting vehicle knows the label value presumably from a prior received detection report. For each classification request received by `uFldHazardSensor`, the simulator makes a classification determination based on the ground-truth declared in the hazard file, and the value of P_c :

- If the object is a hazard, the simulator will report it as a hazard with probability P_c .
- If the object is benign the simulator will report it as benign with probability P_c .

Again, the value of P_c is tied to the prevailing sensor setting chosen for that vehicle. After the classification determination has been made, a report is generated by the simulator to be shared to the vehicle:

```
UHZ_HAZARD_REPORT_ARCHIE = label=12,type=benign
```

Limits the Classification Query Interval

A limit is imposed by `uFldHazardSensor` on the frequency in which classification requests are processed. The default is one classification per 30 seconds. This may be configured differently with the `min_classify_interval` configuration parameter.

A classification request will be honored by `uFldHazardSensor` at any time *after* a detection has been made. Subsequent requests will be honored for each new pass over the object. In other words, `uFldHazardSensor` will not honor multiple classification requests for a single pass over the object.

Managing the Order of Classification Requests

By default, the classify requests are handled in the order in which they are received. This may be overridden to accommodate further consideration of the value of the classification information. The vehicle may associate a *priority* with the classify request, with a posting such as:

```
UHZ_CLASSIFY_REQUEST = vname=archie,label=152,priority=95
```

It is up to the user to determine the appropriate priority, perhaps tying the priority to the expected information gain. The user may also make a request that simply must be the top priority upon arrival. This can be done with a posting such as:

```
UHZ_CLASSIFY_REQUEST = vname=archie,label=152,priority=100,action=top
```

The above request will ensure that object 152 will be processed next. Each time a new entry is pushed into the top of the priority queue in this manner, the clock is reset for that vehicle, and `min_classify_interval` seconds will need to elapse before the next classify result is posted for that vehicle. The FIFO ordering is only relevant for two requests of equal priority, and all requests have a priority of 50 if the priority is left unspecified. One further tool available to the user is to simply clear the queue of pending requests completely:

```
UHZ_SENSOR_CLEAR = vname=archie
```

25.3 Simulator Sensor Configuration Options

The `uFldHazardSensor` simulator needs to be configured with one or more *sensor settings*. A sensor setting is comprised of the following 3-tuple:

- Swath width
- ROC curve exponent
- Classifier constant

These tuples are set in the `uFldHazardSensor` configuration block, and may then be chosen dynamically by the autonomy system on the vehicle by posting to the MOOS variable `UHZ_CONFIG_REQUEST`, by simply naming the requested swath width, and desired probability of detection. By default the hazard sensor limits the frequency of changes to the swath width setting. The default is 300 seconds. This is determined in the `uFldHazardSensor` configuration block with the parameter `min_reset_interval`. The probability of detection, P_D , may be changed as often as desired.

25.3.1 Sensor Swath Width Options

The sensor swath width specifies the width of the sensor field at any given moment, from port to starboard. The sensor range on either side then is simply half the swath width. The swath *length* is set in the `uFldHazardSensor` configuration block and remains the same regardless of the swath width.

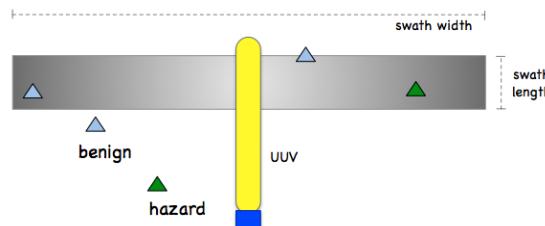


Figure 90: **Sensor Swath Parameter:** The swath width is a parameter that may be re-configured dynamically by the autonomy system. The swath length is set once in the `uFldHazardSensor` configuration block.

25.3.2 Sensor ROC Curve Configuration Options

The ROC curves used in this simulator are based on a simple relationship between probability of detection, P_D , and probability of false alarm, P_{FA} . An example ROC curve is shown in 91. This curve represents

$$P_D = P_{FA}^{0.25}$$

The user of a given sensor with this characteristic ROC curve must decide where to set the detection threshold, P_D . By selection a higher P_D , one must also have to live with a higher P_{FA} . Since the user usually approaches this problem by choosing the P_D , the curve in the figure below could also be described as:

$$P_{FA} = P_D^4$$

The P_{FA} typically follows from a user-chosen P_D . For this reason, the `uFldHazardSensor` simulator is configured by identifying the ROC curve solely by the exponent value in the above function.

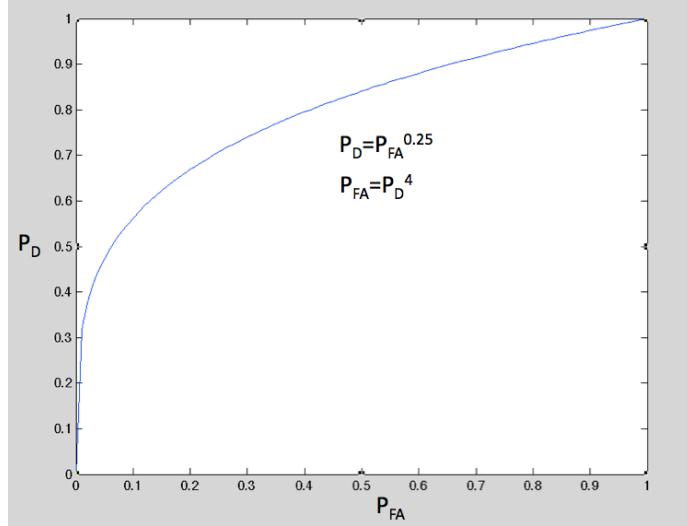


Figure 91: **Example Receiver Operating Characteristic (ROC) Curve:** The ROC curve relates the probability of detection, P_D , on the y-axis, vs. the probability of false alarm, P_{FA} , on the x-axis.

The above ROC may correspond to, for example, the below sensor configuration in the `uFldHazardSensor` configuration block:

```
sensor_config = width=25, exp=4, pclass=0.80
```

The sensor may be configured with more than one ROC curve, typically trading off a more desirable ROC curve for a small sensor swath. For example, the simulated sensor may be configured with the following five options:

```

sensor_config = width=80, exp=2, pclass=0.60
sensor_config = width=65, exp=4, pclass=0.75
sensor_config = width=50, exp=6, pclass=0.85
sensor_config = width=30, exp=12, pclass=0.93
sensor_config = width=18, exp=20, pclass=0.97

```

The idea is shown in Figure 92. When the sensor is configured with these options, it presents the user with *two* choices to make: the sensor width, and choice for P_D . This is done with the `UHZ_CONFIG_REQUEST` interface described in Section 25.3.4.

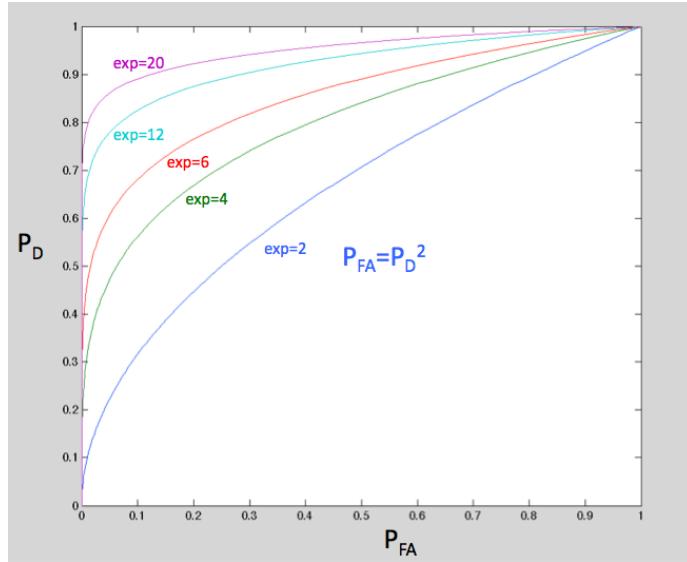


Figure 92: Example ROC Curve Configuration Options: By altering a single component, the exponent component of the expression $P_{FA} = P_D^{exp=2,4,6,12,20}$, the ROC curve characteristics may be varied from least desirable ($exp = 1$) to most desirable ($exp = 20$ or higher).

If the sensor user (a given vehicle) does not request a particular swath width and P_D , the simulator will choose one of the options as a default. The default swath width is selected by choosing the highest width that is not more than the average between the highest and lowest width. In the settings above, for example, the `width=30` is the highest width not greater than the average of the two extremes (49). The default P_D setting is 0.9 if left unspecified.

25.3.3 Classification Configuration Options

Classification refers to the process of handling a detected object and determining if it is either a hazard or a benign object (false alarm), described in Section 25.2.2. The classification stage is shown in Figure 84. In the simulator, this is simply implemented by rolling the dice according to a single probability metric P_C . This is the probability that the classification determination is correct. Since $P_C = 1$ and $P_C = 0$ provide the same net utility, the range of values for P_C is from $[0.5, 1]$. This value is tied to a particular swath width and ROC curve characteristic. The idea is that a smaller swath width allows the sensor to provide a denser set of image data for a smaller area, which leads

to both the better ROC curve as well as crisper images for a classification algorithm (even if the classifying agent is a human examining images off-board).

25.3.4 Dynamic Resetting of the Sensor

The `uFldHazardSensor` simulator is configured with a set of configuration options, as described above, of which one must be chosen, along with a chosen P_D . This is done by the vehicle posting a configuration request of the form:

```
UHZ_CONFIG_REQUEST = vname=archie, width=32, pd=0.95
```

The first field tells the simulator which vehicle is making the request. The second field identifies the ROC curve and swath width. The last field identifies the point on the ROC curve and thus determines the P_{FA} . The `uFldHazardSensor` simulator processes this request by matching up the request to the set of possible options. If the exact swath width requested is not available, the next lowest is chosen. If the requested swath width is less than the lowest option, the lowest option is chosen. The simulator sends an acknowledgment to the vehicle in the form of:

```
UHZ_CONFIG_ACK_ARCHIE = width=30,pd=0.9,pfa=0.28,pclass=0.93
```

The sharing of the variable `UHZ_CONFIG_ACK_ARCHIE` between the shoreside and the vehicle archie is handled by `pShare`, typically with dynamic registrations automatically configured by `uFldShoreBroker`. By default the hazard sensor limits the frequency of changes to the sensor settings. The default is 300 seconds. This is determined in the `uFldHazardSensor` configuration block with the parameter `min_reset_interval`. This only restricts how often the swath width may be reset. The desired value for the probability of detections, P_D , may be reset as often as desired. In this case the configuration request would simply not specify a desired swath width. For example:

```
UHZ_CONFIG_REQUEST = vname=archie, pd=0.75
```

25.3.5 Posting of Sensor Configuration Options

The sensor configuration options may be known to other MOOS processes by subscribing to the MOOS variable, `UHZ_OPTIONS_SUMMARY`. The following would be published:

```
UHZ_OPTIONS_SUMMARY = width=25,exp=4,pclass=0.85:width=50,exp=2,pclass=0.60 \
width=10,exp=6,pclass=0.93
```

corresponding to the following configuration of `uFldHazardSensor`:

```
sensor_config = width=25, exp=4, pclass=0.85
sensor_config = width=50, exp=2, pclass=0.60
sensor_config = width=10, exp=6, pclass=0.93
```

Since this information rarely if ever changes after its first posting, it is not posted very often. By default it is posted once every ten seconds. Since this posting is typically occurring on a shoreside computer, bridged to a remote vehicle, a long delay may not be desirable. This interval may be altered, for example to 5 seconds, by configuring `options_summary_interval=5`.

25.4 Configuring the Hazard Field

The hazard field is configured either by reading in a hazard file, or by explicitly listing the hazards in the `uFldHazardSensor` configuration block. The former method is recommended since the hazard file is typically also read by `uFldHazardMetric` for grading hazardset reports against the ground truth. The ground truth obviously needs to be the same between applications and this is easier to ensure if they are just pointing to the same file. Either way, the format for specifying an object is the same and has the following fields by example:

```
hazard = x=98.2,y=-127.7,label=14,type=benign
```

Below is a list of possible fields for specifying an object. The first four fields, used in the example above are mandatory. Uniqueness between labels is mandatory as well.

- [x] position
- [y] position
- [type] (hazard or benign)
- [label] (unique between entries)

An optional field for specifying a *hazard resemblance* factor, discussed further in Section ??:

- [hr]: a value in the range [0, 1] associated with benign objects

Optional fields for specifying an optimal aspect angle for correctly detecting and classifying an object, discussed further in Section ??:

- [aspect]: the optimal sensing aspect angle
- [aspect_min]: the aspect angle range within which sensing is still optimal
- [aspect_max]: the aspect angle range beyond which sensing is fully degraded

Optional fields for specifying rendering hints on a per-object basis:

- [color] (rendering hint, default: green for hazards, light_blue for benign objects)
- [shape] (rendering hint, default: triangle for both hazards and benign objects)
- [width] (rendering hint, default: 8 meters for both hazards and benign objects)

So an object specification utilizing all object parameters may look like:

```
hazard = x=98.2,y=-127.7,label=14,type=benign,hr=0.8,aspect=43,aspect_min=30,  
aspect_max=60,color=yellow,shape=square,width=4
```

25.4.1 An Example Hazard Field

The field in Figure 93 below shows an example field. This is also the hazard field used in the example mission, `m10-jake`, described briefly in Section 25.1.1, and more fully in Section 25.12.

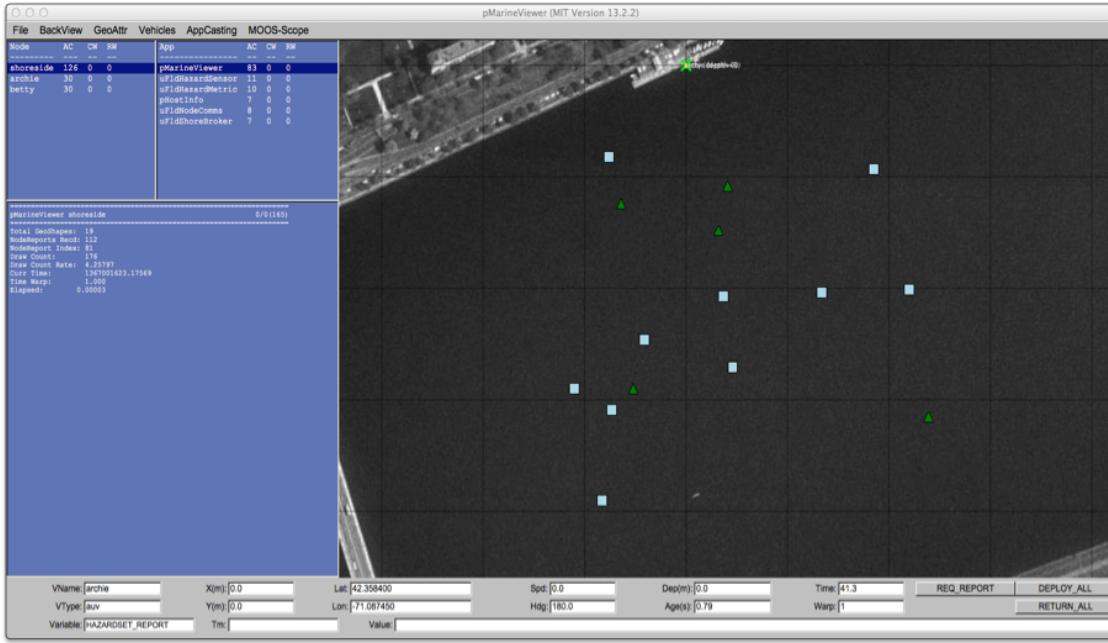


Figure 93: **Simulated Hazard Field:** A hazard field with 18 objects is shown, some are hazardous objects, some are benign objects.

The hazard field may be configured with either entries in the `uFldHazardSensor` configuration block, or by reading in a hazard field configuration file. The format for a configuration line is the same in both cases. For the hazard field shown in Figure 93, the configuration lines used were:

```

hazard = x=-52,y=-290,label=75,type=hazard
hazard = x=41,y=-108,label=53,type=hazard
hazard = x=-64,y=-124,label=98,type=hazard
hazard = x=32,y=-148,label=91,type=hazard
hazard = x=239,y=-315,label=19,type=hazard
hazard = x=-41,y=-246,label=82,type=benign
hazard = x=185,y=-93,label=64,type=benign
hazard = x=-110,y=-290,label=11,type=benign
hazard = x=-76,y=-82,label=14,type=benign
hazard = x=-73,y=-309,label=28,type=benign
hazard = x=46,y=-271,label=73,type=benign
hazard = x=-83,y=-390,label=77,type=benign
hazard = x=220,y=-201,label=15,type=benign
hazard = x=134,y=-204,label=95,type=benign
hazard = x=37,y=-207,label=99,type=benign

```

25.4.2 Automatically Generating a Hazard Field

There also exists in the moos-ivp tree a small utility call `gen_hazards`. This simple command line tool will generate a list of randomly generated objects of a specified type, given a convex polygon as input. For example:

```
$ gen_hazards --polygon=-150,-75:-150,-400:400,-400:400,-75 --objects=5,hazard  
--objects=10,benign  
hazard = x=-52,y=-290,label=75,type=hazard  
hazard = x=41,y=-108,label=53,type=hazard  
hazard = x=-64,y=-124,label=98,type=hazard  
hazard = x=32,y=-148,label=91,type=hazard  
hazard = x=239,y=-315,label=19,type=hazard  
hazard = x=-41,y=-246,label=82,type=benign  
hazard = x=185,y=-93,label=64,type=benign  
hazard = x=-110,y=-290,label=11,type=benign  
hazard = x=-76,y=-82,label=14,type=benign  
hazard = x=-73,y=-309,label=28,type=benign  
hazard = x=46,y=-271,label=73,type=benign  
hazard = x=-83,y=-390,label=77,type=benign  
hazard = x=220,y=-201,label=15,type=benign  
hazard = x=134,y=-204,label=95,type=benign  
hazard = x=37,y=-207,label=99,type=benign
```

If the information is stored in file, then `uFldHazardSensor` may then be configured with the single line `hazard_file=hazards.txt`. This utility also supports command line options for randomly generating hazard resemblance factors and aspect information for each object.

25.5 Hazard Resemblance Factor

The simulator may be configured with benign objects that have an optional *resemblance factor*. This is a number in the range $[0, 1]$, where zero indicates the object looks absolutely nothing like a hazard, and one indicates a very close resemblance to a hazard. When this factor is present, it affects both the probability of detection, P_D , and the probability of classification, P_C . Such objects are configured like any other object, but have an additional parameter, for example:

```
hazard = x=37,y=-207,label=99,type=benign,hr=0.73
```

25.5.1 The Effect of the Resemblance Factor on Detections

The general detection algorithm was described in Section 25.2.1. In short, hazards generate detections with probability P_D , and benign objects (falsely) generate detections with probability P_{FA} . When a benign object has a hazard resemblance factor, R , the probability of false alarm is instead $(P_{FA} + R)/2$. Since the R factor has a range of $[0, 1]$, the modified probability also still has a range of $[0, 1]$.

25.5.2 The Effect of the Resemblance Factor on Classifications

As with detections, the hazard resemblance factor only affects the probability of correct classifications for benign objects. If the object is benign and *does* have an associated hazard resemblance factor,

R , the simulator will report it as benign with probability $P_C + (1 - P_C)(1 - R)$. In other words, if a benign object happens to have a very high resemblance to a hazard, $R = 1$, the probability of a correct classification is at its lowest, P_C . If the benign objects happens to have very little resemblance to a hazard, $R = 0$, the probability of classification is at its highest, nearly 1.

25.5.3 Generating a Random Hazard Field with Resemblance Factors

A hazard resemblance factor may be automatically generated with the `gen_hazards` utility by including an optional command-line argument, `--exp=N`, where N is in the range [0.01, 10]. When this argument is provided, the utility starts by generating, for each benign object, a resemblance factor over the range [0, 1] with uniform probability. It then raises this value to the number provided by the `--exp` parameter. The higher this parameter, the lower the expected value for the resulting resemblance factors. For example, with `--exp=1`:

```
gen_hazards --polygon=-150,-75:-150,-400:400,-400:400,-75 --objects=4,benign --exp=1
hazard = x=149,y=-216,label=82,type=benign,hr=0.605
hazard = x=263,y=-88,label=51,type=benign,hr=0.874
hazard = x=292,y=-185,label=96,type=benign,hr=0.306
hazard = x=-57,y=-194,label=91,type=benign,hr=0.07
hazard = x=-144,y=-330,label=58,type=benign,hr=0.801
```

With `--exp=10`:

```
gen_hazards --polygon=-150,-75:-150,-400:400,-400:400,-75 --objects=4,benign --exp=10
hazard = x=390,y=-137,label=39,type=benign,hr=0.00001
hazard = x=236,y=-213,label=46,type=benign,hr=0.12772
hazard = x=172,y=-215,label=76,type=benign,hr=0.01033
hazard = x=-114,y=-235,label=8,type=benign,hr=0.40688
```

With `--exp=0.01`:

```
gen_hazards --polygon=-150,-75:-150,-400:400,-400:400,-75 --objects=4,benign --exp=0.01
hazard = x=226,y=-379,label=85,type=benign,hr=0.96231
hazard = x=231,y=-167,label=81,type=benign,hr=0.92962
hazard = x=108,y=-101,label=74,type=benign,hr=0.78424
hazard = x=225,y=-379,label=86,type=benign,hr=0.9767
```

25.6 Aspect Angle Sensitivity

The simulator may be configured with objects having an *aspect sensitivity*. In this case, a unique optimal aspect angle may be associated with each object. Depending on the vehicle's approach angle to the object, the sensor performance may be degraded the further the approach angle deviates from the optimal aspect angle. Such objects are configured like any other object, but have three additional parameters, for example:

```
hazard = x=37,y=-207,label=99,type=benign,aspect=57,aspect_min=20,aspect_max=45
```

The idea is that the vehicle heading, as its sensors pass over the object, is relevant to the sensor performance. If the vehicle heading matches the object's optimal aspect angle, the sensor data collected is optimal with respect to the simulated sensor processing algorithms. The more the vehicle approach heading differs from the optimal aspect angle, the more the sensor is degraded in terms of P_D , P_{FA} , and P_C .

To be more precise, a sensor `degradation` value is calculated, in the range of [0, 1], i.e., zero to one hundred percent. This value is determined by first calculating the `heading_delta`, representing difference in vehicle approach heading compared to the object's optimal aspect angle. This value is in the range [0, 90], with zero meaning the vehicle approach heading was dead-on perfect with the object's optimal aspect angle. The simulator knows the `vehicle_heading` from incoming node reports, and the optimal aspect angle, `aspect`, `aspect_min`, and `aspect_max` for an object based on the ground truth hazard file. First, the `heading_delta` is determined by:

$$\text{heading_delta} = \min \begin{cases} (|\text{vehicle_heading} - \text{aspect}|) \bmod 180 \\ 180 - ((|\text{vehicle_heading} - \text{aspect}|) \bmod 180) \end{cases} \quad (9)$$

Then, the sensor `degradation` is determined by:

$$\text{degradation} = \begin{cases} 0 & \text{heading_delta} \leq \text{aspect_min} \\ k & \text{aspect_min} < \text{heading_delta} \leq \text{aspect_max} \\ 100 & \text{aspect_max} < \text{heading_delta} \end{cases} \quad (10)$$

$$k = (\text{heading_delta} - \text{aspect_min}) / (\text{aspect_max} - \text{aspect_min}) \quad (11)$$

The degradation value is used to modify the detection and classification algorithms, described next.

25.6.1 The Effect of Aspect Sensitivity on the Detection Algorithm

The general detection algorithm was described in Section 25.2.1. In short, hazards generate detections with probability P_D , and benign objects (falsely) generate detections with probability P_{FA} . The relationship between P_D and P_{FA} is based on the ROC curve associated with the prevailing sensor settings. Aspect sensitivity is applied in a two step process:

1. Determination of sensor `degradation` as described above, and
2. Application of sensor degradation based on a modified ROC curve.

A full (one hundred percent) sensor degradation essentially is equivalent to the sensor working with a ROC curve of exponent 1, a linear line, as discussed in Section 25.3.2, and in Figure 91. In this case, P_D and P_{FA} are equivalent at all points on the curve; a detection event provides no insight into the object type. When there is no sensor degradation, the form of the ROC curve remains unchanged. In between, with partial sensor degradation, the ROC curve is somewhere in between its original form and 1 (linear). For example, if the original ROC curve was formed with an exponent of 5, 50% sensor degradation entails a ROC curve of 3 instead. If the user had chosen a P_D of 0.9, the corresponding P_{FA} would have been $(0.9)^5 = 0.59$ but is instead $(0.9)^3 = 0.73$ with 50% degradation.

25.6.2 The Effect of Aspect Sensitivity on the Classification Algorithm

The general classification algorithm was described in Section 25.2.2. In short, each time a classification request is processed, the probability of a correct classification is P_C . This holds whether the actual object is a hazard or benign object. Aspect sensitivity is applied in a two step process:

1. Determination of sensor `degradation` as described above, and
2. Application of sensor degradation to modify the probability of classification to this object on this pass.

A full sensor degradation is equivalent to setting $P_C = 0.5$, which is essentially as good as a random coin flip. For partial degradations, the P_C is revised downward:

$$P'_C = 0.5 + (1 - \text{degradation}) * (P_C - 0.5)$$

25.6.3 Visual Cues on Aspect Sensitivity

Normally `uFldHazardSensor` generates visual cues upon detections and classifications; white circles on a detection, yellow or green pentagons upon classification reports. To aid things further, the application calculates detection and classification results with and without aspect sensitivity. When the sensor degradation is noted to have an adverse effect, the interior of the circles or polygons are filled in with a pink color rather than their normal white, green or yellow color.

25.7 Configuring the Simulator Visual Preferences

As shown in Figure 86, the `uFldHazardSensor` is typically running in the shoreside community, alongside a GUI application like `pMarineViewer`. Certain messages are posted by `uFldHazardSensor` for visualization by default. They may be shut off, or have their properties altered by configuring the simulator as desired.

25.7.1 Configuring the Sensor Field Swath Rendering

The sensor field is normally rendered as a rectangle that moves along with the vehicle. This rectangle is drawn to exactly correspond to the sensor field. Rendering in `pMarineViewer` is accomplished by a posting of `VIEW_POLYGON` by `uFldHazardSensor`. This may be shut off with the following configuration:

```
show_swath = false
```

The swath is slightly transparent to allow for the objects to be seen under the swath. The default transparency is 0.2, but may be changed with the following configuration:

```
swath_transparency = 0.60
```

While the length of the rendered swath width from side to side is determined by the sensor setting, the length of the rendered swath width from front to back is determined by the configuration:

```
swath_length = 5 // In meters. 5 is the default.
```

25.7.2 Configuring the Hazard Field Renderings

Rendering of the hazard field is done at the start of `uFldHazardSensor` by producing a `VIEW_MARKER` posting once for each object in the field. This may be disabled with:

```
show_hazards = false
```

Although each object in the hazard file may be configured with color, shape, and width, the default values may be provided for any object that leaves any one of these fields unspecified.

```
default_hazard_shape = triangle
default_hazard_color = green
default_hazard_width = 8
default_benign_shape = triangle
default_benign_color = green
default_benign_width = 8
```

25.7.3 Configuring the Sensor Report Renderings

Sensor reports are rendered as circles around the objects in the hazard field. If a detection is not made, no circle is rendered. If a detection is made and classified as a hazard, a yellow pentagon is rendered. If classified as benign, a green pentagon is rendered. All such renderings are accomplished by `uFldHazardSensor` making a posting to `VIEW_CIRCLE`, then processed by `pMarineViewer`.

When multiple vehicles are using the simulator, it may be confusing after some time to know which vehicle was responsible for which object rendered. For this reason, the `uFldHazardSensor` may be configured to have the detection circles disappear after some number of seconds. This can be done with:

```
show_detections = 60
```

The above will result in each detection circle being rendered for 60 seconds before disappearing from view.

25.7.4 Rendering the Prevailing P_D and P_{FA} Values

Normally the prevailing P_D and P_{FA} values are rendered alongside the vehicle's sensor swath as shown in Figure 94. These may be either configured on or off with the following two configuration parameters:

```
show_pd = true // the default is true
show_pfa = true // the default is false
```

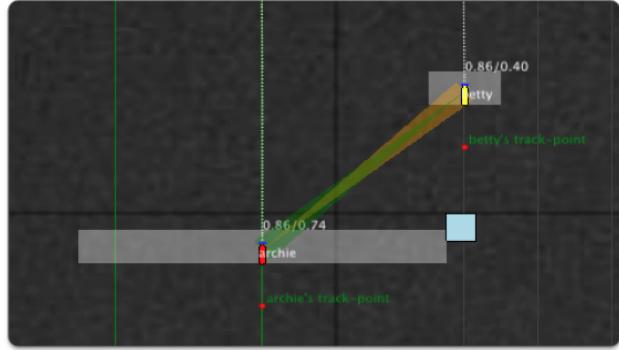


Figure 94: The prevailing P_D and P_{FA} values may be optionally rendered alongside the sensor swath.

25.8 Under the Hood: Sensor Blackouts During Turns

Image data collected during turns is notoriously poor, effectively rendering that data useless for automatic image processing algorithms. The hazard simulator therefore has provisions to disable itself during turns. It does so by storing a recent vehicle heading history from node reports, and disabling detections during turns. The visual aspects of the sensor swath also change as shown in Figure 95.

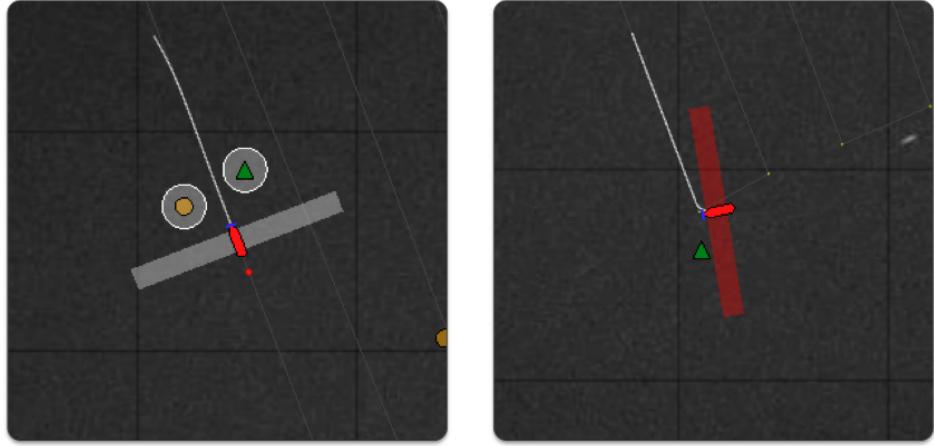


Figure 95: **Sensor blackouts during turns:** A vehicle will render its swath in white when running straight and on, and will render the swath in red when turning and disabled. In the image on the right, the vehicle may not have detected the object it just passed even if it were not turning, but in this case the lack of detection was ensured due to the turning of the vehicle.

By default the sensor will be disabled when the recent turn rate (defined over the previous 2 seconds) exceeds a rate of 1.5 degrees per second. This may be adjusted in the mission file with the following parameter:

```
max_turn_rate = 3.0
```

25.9 Configuration Parameters of uFldHazardSensor

The following parameters are defined for `uFldHazardSensor`. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so.

Listing 25.2: Configuration Parameters for `uFldHazardSensor`.

<code>default_benign_color:</code>	Color for rendering benign objects. Legal values: any color in Appendix B. The default is lightblue. Section 25.7.2.
<code>default_benign_shape:</code>	Shape for rendering benign objects Legal values: <i>square</i> , <i>circle</i> , <i>diamond</i> , <i>triangle</i> . The default is <i>triangle</i> . Section 25.7.2.
<code>default_benign_width:</code>	Width in meters for rendering benign objects. The default is 8. Section 25.7.2.
<code>default_hazard_color:</code>	Color for rendering hazard objects. Legal values: any color in Appendix B. The default is green. Section 25.7.2.
<code>default_hazard_shape:</code>	Shape for rendering hazard objects. Legal values: <i>square</i> , <i>circle</i> , <i>diamond</i> , <i>triangle</i> . The default is <i>triangle</i> . Section 25.7.2.
<code>default_hazard_width:</code>	Width in meters for rendering hazard objects. The default is 8. Section 25.7.2.
<code>hazard_file:</code>	Names a file containing the hazard field configuration. Section 25.4.
<code>max_turn_rate:</code>	Maximum rate of turn in meters per second, above which the sensor will not produce detections. The default is 1.5. Section 25.8.
<code>min_classify_interval:</code>	Minimum amount of time between classification results per vehicle. The default is 30 seconds. Section 25.2.2.
<code>min_reset_interval:</code>	Minimum amount of time between sensor configuration resets per given vehicle. The default is 300 seconds. Section 25.3.
<code>options_summary_interval:</code>	Duration (secs) between posting of options summary. The default is 10. Section 25.3.5.
<code>seed_random:</code>	If true, a random number generated is seeded. Legal values: true, false. The default is true.
<code>sensor_config:</code>	Describes one of possibly many sensor config options. Section 25.3.
<code>show_detections:</code>	Duration attached to detection circle postings. Legal values: any numerical value or the keyword "nolimit". The default is nolimit. Section 25.7.3.
<code>show_hazards:</code>	If false, hazard field visuals are not posted. Legal values: true, false. The default is true. Section 25.7.2.
<code>show_pd:</code>	If true, the probability of detection, P_D , associated with a particular vehicle will be rendered alongside the swath rendering. Legal values: true, false. The default is true. Section 25.7.4.
<code>show_pfa:</code>	If true, the probability of false alarm, P_{FA} , associated with a particular vehicle will be rendered alongside the swath rendering. Legal values: true, false. The default is false. Section 25.7.4.

- `show_swath`: If false, vehicle sensor swath visuals are not posted. Legal values: true, false. The default is true. Section [25.7.1](#).
- `swath_transparency`: Transparency used for rendering swath. Legal values: [0, 1]. The default is 0.2. Section [25.7.1](#).
- `swath_length`: Extent of the sensor swath, in meters, in the direction of bow to stern. Legal values: any numerical value. Values less than 1 will be clipped to 1. The default is 5.

25.9.1 An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command-line:

```
$ uFldHazardSensor --example or -e
```

This will show the output shown in Listing 3 below.

Listing 25.3: Example configuration for `uFldHazardSensor`.

```

1 =====
2 uFldHazardSensor Example MOOS Configuration
3 =====
4
5 ProcessConfig = uFldHazardSensor
6 {
7     AppTick      = 4
8     CommsTick   = 4
9
10    // Configuring visual preferences
11    default_hazard_shape = triangle           // default
12    default_hazard_color = green              // default
13    default_hazard_width = 8                 // default
14
15    default_benign_shape = triangle           // default
16    default_benign_color = light_blue         // default
17    default_benign_width = 8                 // default
18    swath_transparency = 0.2                // default
19
20    sensor_config = width=25, exp=4, class=0.80
21    sensor_config = width=50, exp=2, class=0.60
22    sensor_config = width=10, exp=6, class=0.93,max=1
23    hazard_file   = hazards.txt
24    swath_length  = 5                      // default
25    seed_random   = false                  // default
26
27    show_hazards  = true // default        // default
28    show_swath    = true // default        // default
29    show_detections = 60 // seconds (unlimited if unspecified)
30    show_pd       = true // pd shown with swaths // default
31    show_pfa      = true // pfa shown with swaths // default
32
```

```

33     min_reset_interval      = 300 // default
34     min_classify_interval   = 30  // default
35     options_summary_interval = 10 // default
36 }

```

25.10 Publications and Subscriptions for uFldHazardSensor

The interface for `uFldHazardSensor`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldHazardSensor --interface or -i
```

25.10.1 Variables Published by uFldHazardSensor

The primary output of `uFldHazardSensor` to the MOOSDB is posting of sensor reports, visual cues for the sensor reports, and visual cues for the hazard objects themselves.

- `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 25.11
- `UHZ_DETECTION_REPORT`: A report on a detection made by the sensor for an object in the hazard field. It includes the name of the vehicle.
- `UHZ_DETECTION_REPORT_NAMEJ`: A report on a detection made by the sensor (on vehicle `NAMEJ`) for an object in the hazard field.
- `UHZ_HAZARD_REPORT`: A report on a classification made by the sensor for an object in the hazard field. It includes the name of the vehicle.
- `UHZ_HAZARD_REPORT_NAMEJ`: A report on a classification made by the sensor (on vehicle `NAMEJ`) for an object in the hazard field.
- `UHZ_OPTIONS_SUMMARY`: A report the possible sensor settings available to the user.
- `UHZ_CONFIG_ACK`: An acknowledgment message sent to the vehicle verifying a requested sensor setting.
- `VIEW_CIRCLE`: A visual artifact for rendering a circle around a hazard, indicating the detection and classification.
- `VIEW_MARKER`: A visual artifact for rendering objects in the hazard field.
- `VIEW_POLYGON`: A visual artifact for rendering a rectangle around a vehicle, indicating a vehicle sensor field moving with the vehicle.

Example postings:

```

UHZ_DETECTION_REPORT      = vname=archie,x=51,y=11.3,label=12
UHZ_DETECTION_REPORT_ARCHIE = x=51,y=11.3,label=12
UHZ_HAZARD_REPORT         = vname=archie,x=51,y=11.3,hazard=true,label=12
UHZ_HAZARD_REPORT_ARCHIE = x=51,y=11.3,hazard=true,label=12
UHZ_CONFIG_ACK            = vname=archie,width=20,pd=0.9,pfa=0.53,pclass=0.91
UHZ_OPTIONS_SUMMARY       = width=10,exp=6,pclass=0.9:width=25,exp=4,pclass=0.85

```

The vehicle name may be embedded in the MOOS variable name to facilitate distribution of report messages to the appropriate vehicle with `pShare`.

25.10.2 Variables Subscribed for by uFldHazardSensor

The `uFldHazardSensor` application will subscribe for the following four MOOS variables:

- `APPCAST_REQ`: A request to generate and post a new appcast report, with reporting criteria, and expiration. Section 11.10.4.
- `UHZ_SENSOR_REQUEST`: A message from the vehicle indicating the sensor is active.
- `UHZ_SENSOR_CONFIG`: A message from the vehicle requesting one of the possible sensor settings and P_D choice from the ROC curve resulting from the sensor settings.
- `NODE_REPORT`: A report on a vehicle location and status.
- `NODE_REPORT_LOCAL`: A report on a vehicle location and status.

Example postings

```
UHZ_SENSOR_REQUEST = vname=archie
UHZ_CONFIG_REQUEST = vname=archie,width=50,pd=0.9
```

Command Line Usage of uFldHazardSensor

The `uFldHazardSensor` 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:

```
$ uFldHazardSensor --help or -h
```

This will show the output shown in Listing 4 below.

Listing 25.4: Command line usage for the `uFldHazardSensor` tool.

```
1 =====
2 Usage: uFldHazardSensor file.moos [OPTIONS]
3 =====
4
5 Options:
6   --alias=<ProcessName>
7     Launch uFldHazardSensor with the given process
8     name rather than uFldHazardSensor.
9   --example, -e
10    Display example MOOS configuration block.
11   --help, -h
12    Display this help message.
13   --interface, -i
14    Display MOOS publications and subscriptions.
15   --version,-v
```

```

16      Display release version of uFldHazardSensor.
17      --verbose=<setting>
18          Set verbosity. true or false (default)
19
20 Note: If argv[2] does not otherwise match a known option,
21       then it will be interpreted as a run alias. This is
22       to support pAntler launching conventions.

```

25.11 Terminal and AppCast Output

The `uFldHazardSensor` application produces some useful information to the terminal and identical content through appcasting. An example is shown in Listing 5 below. On line 2, the name of the local community, typically the shoreside community, is listed on the left. On the right, "0/0(457)" indicates there are no configuration or run warnings, and the current iteration of `uFldHazardSensor` is 457. Lines 4-6 show the name of the ground truth hazard file and the number of hazards and benign objects.

Lines 8-16 convey the available sensor configuration options, set with the `sensor_config` parameter.

Listing 25.5: Example `uFldHazardSensor` console output.

```

1 =====
2 uFldHazardSensor shoreside                               0/0(457)
3 =====
4 Hazard File: (hazards.txt)
5     Hazard: 11
6     Benign: 8
7
8 =====
9 Sensor Configuration Options
10 =====
11 Width Exp Classify
12 ---- - -
13 10.0   6.0   0.930
15 425.0   4.0   0.850
16 50.0   2.0   0.600
17
18 =====
19 Sensor Settings / Stats for known vehicles:
20 =====
21 Vehicle   Swath           Sensor  Sensor
22 Name      Width  Pd    Pfa   Pclass Resets Requests Detects
23 -----  -----  -----  -----  -----  -----  -----  -----
24 archie    50.0   0.860  0.740  0.600  1(12)   203      0
25
26 =====
27 Most Recent Events (1):
28 =====
29 [11.86]: Setting sensor settings for: archie

```

25.12 The Jake Example Mission Using uFldHazardSensor

The *Jake* mission is distributed with the MOOS-IvP source code and contains a ready example of the **uFldHazardSensor** application, configured with hazard field in an included text file. Assuming the reader has downloaded the source code available at www.moos-ivp.org and built the code according to the discussion in Section 1.2.9, the *Jake* mission may be launched by:

```
$ cd moos-ivp/ivp/missions/m10_jake/  
$ ./launch.sh 10
```

The argument, 10, in the line above will launch the simulation in 10x real time. Once this launches, the **pMarineViewer** GUI application should launch and the mission may be initiated by hitting the DEPLOY button.

25.12.1 What is Happening in the Jake Mission

The Jake mission is comprised of two simulated vehicles, *archie* and *betty*. There are three MOOS communities launched, one for the shoreside and one each for the two vehicles. See Figure 86. The **uFldHazardSensor** simulator is running in the shoreside community. The Jake mission is comprised of two phases, the *broad-area-search* phase and the *reacquire* phase. In this mission, archie handles the first phase, passes his results to betty, who handles the second phase.

The Broad Area Search Phase

In the *broad-area-search* phase, a search region is given to archie, in which it is to search for a set of objects, some of which may be hazardous. Knowing nothing a priori about the location of the objects, only the region containing them, archie executes a lawnmower search pattern over this area, as shown in Figure 96. The snapshot in the figure depicts archie having executed most of its pattern, to the West proceeding East. The hazard field is rendered with actual hazards drawn in green triangles, and benign objects drawn in light blue squares. The circles represent detections reported by **uFldHazardSensor**. The yellow circles represent objects classified as hazards, and the white circles represent objects classified as benign.

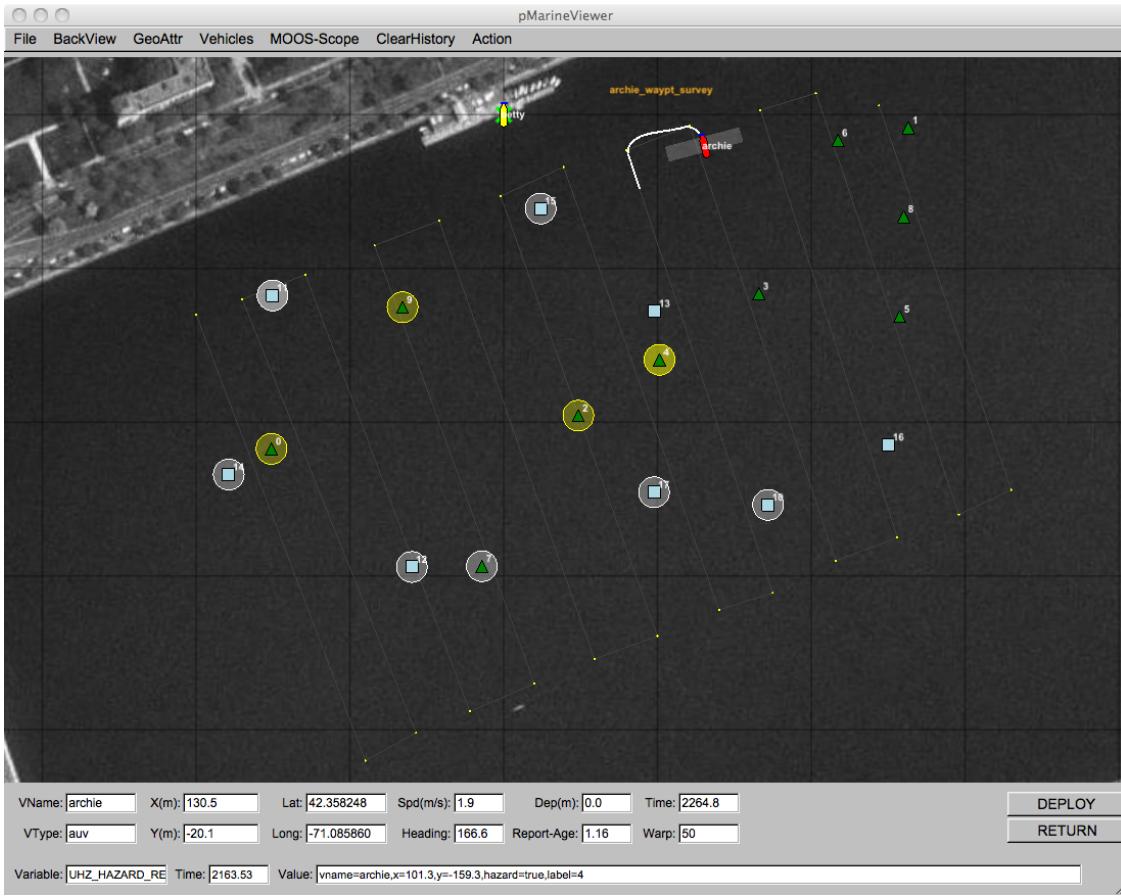


Figure 96: **Simulated Hazard Sensor:** A vehicle processes a series of sensor images which may or may not contain an object. The detection algorithm processes the image and rejects images it believes does not contain hazardous objects. It passes on images containing possible hazardous objects to a classifier, which makes a determination for each object in an incoming image as to whether or not the object is hazardous or benign.

In this mission, the vehicle's sensor is configured with a swath width of 25 meters (12.5 on either side), a probability of detection $P_D = 0.9$, probability of false alarm $P_{FA} = 0.66$, and probability of correct classification $P_C = 0.85$. Note in the above snapshot, the vehicle has successfully detected all hazards, but also detected all but one benign object. It has correctly classified all but one object.

The Reacquire Phase

In the *reacquire* phase, the archie vehicle has returned to the dock, and betty vehicle has been a mission to revisit a set of points. In this case betty has an idea where those points lie and is following a simple path, as shown in Figure 97. Presumably the list of objects to visit and their locations have been communicated to betty from archie. (In this mission things were hard-coded, no message passing actually occurred.) The objective of betty is to use a sensor with a smaller swath width and better sensor processing algorithm to reduce the classification uncertainty associated with the objects being visited.

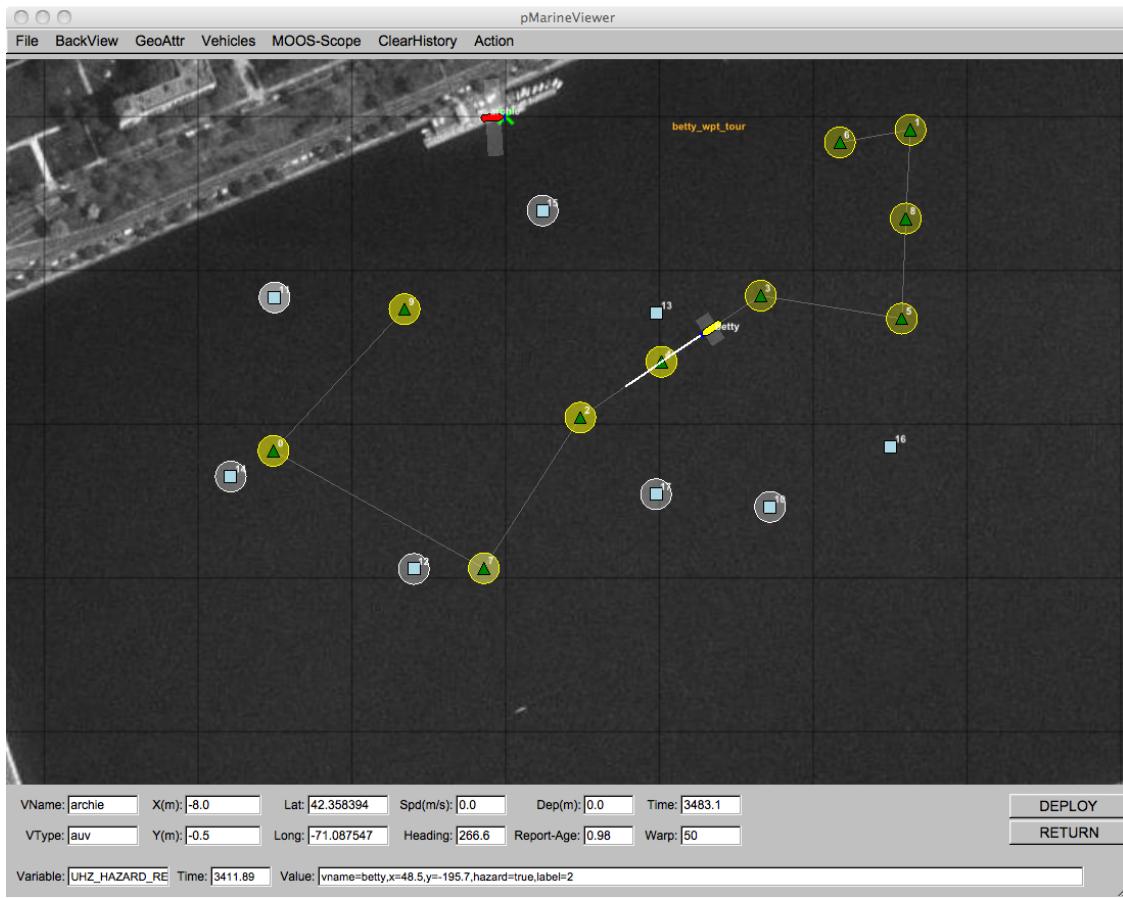


Figure 97: **A Reacquire Mission:** A second vehicle revisits a set of previously detected objects and seeks to reduce the uncertainty associated with their initial classification made with a less reliable sensor.

In this mission, the vehicle's sensor is configured with a swath width of 10 meters (5 meters on either side), a probability of detection $P_D = 0.9$, probability of false alarm $P_{FA} = 0.53$, and probability of correct classification $P_C = 0.93$.

26 uFldHazardMgr: On-Board Management of a Hazard Sensor

26.1 Overview

The **uFldHazardMgr** application is *straw man* module for interacting with an on-board hazard sensor. It does two basic things as implied in Figure 98:

1. *Interacts with the sensor*: It decides a sensor configuration setting and sends this to the hazard sensor. It may change settings during the course of the mission to its advantage. It interacts with the sensor by sending sensor requests, and receiving sensor reports. It has some knowledge of the sensor properties and configuration options.
2. *Generates a hazardset report*: It builds an internal belief state regarding the identification and location of hazards, and reports this belief state upon request.

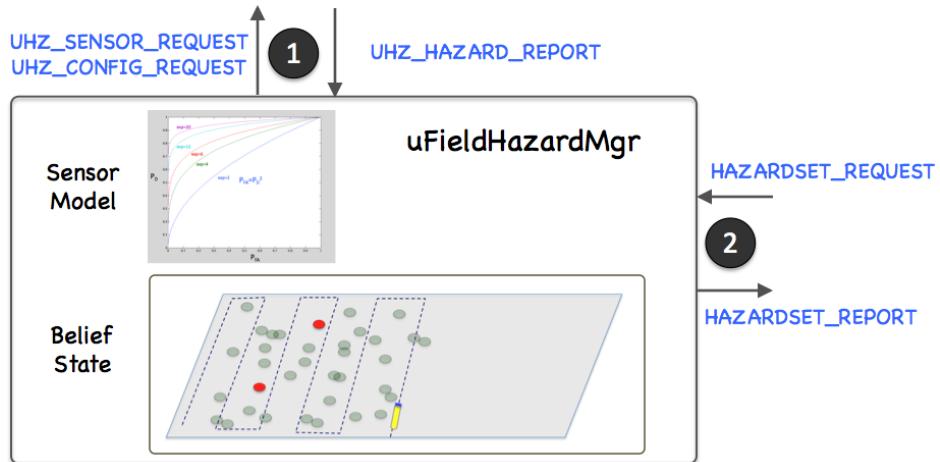


Figure 98: **The uFldHazardMgr**: interacts with the on-board sensor and processes sensor information and generates a report, upon request, regarding the identification and location of hazards. The arrows indicate the key MOOS variables used for interacting with the sensor and generating reports.

There are many ways to interpret sensor data and ultimately decide upon a hazardset report. For this reason we regard **uFldHazardMgr** as a *straw man* module. It implements the key syntactic steps to minimally configure and interact with the sensor and produce a syntactically correct hazardset report. The intention is that users may wish to use this as a starting point.

A further aspect one may wish to include in this module would be a form of reasoning about the vehicle's path through the hazard field. This module, and accompanying example mission, was written with the vehicle simply performing a lawnmower exhaustive search through the field. This module could be used in conjunction with the helm to decided follow-up search patterns or collaborative strategies with other vehicles. Again, this is not part of **uFldHazardMgr** since the objective of this module is to provide a syntactically valid starting point for managing sensor information.

26.2 Using uFldHazardMgr

Typical Simulator Topology

The typical module topology is shown in Figure 99 below. The `uFldHazardMgr` is situated in the vehicle MOOS community. It interacts with the `uFldHazardSensor` situated in the shoreside MOOS community. The vehicle communicates with the shoreside community using `pShare`. The shoreside knows the location of all vehicles from node reports received from each vehicle running `pNodeReporter`. The `uFldHazardMgr` first initializes the sensor by sending a configuration request via `UHZ_CONFIG_REQUEST`. The hazard sensor acknowledges the configuration with a `UHZ_CONFIG_ACK` message. Thereafter the hazard manager may interact with the sensor by sending sensor requests with `UHZ_SENSOR_REQUEST` and periodically receiving reports of detections with `UHZ_DETECTION_REPORT`.

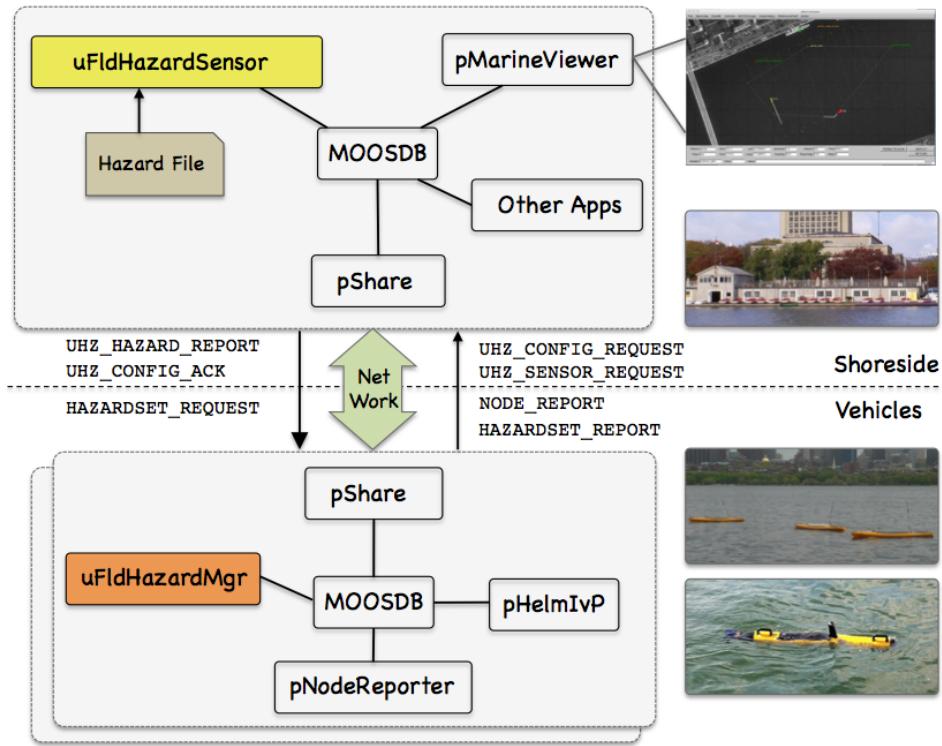


Figure 99: **Typical uFldHazardMgr Topology:** The simulator runs in a shoreside computer MOOS community and is configured with a hazard field containing both hazards and benign objects. Vehicles accessing the simulator send a steady stream of messages (`UHZ_SENSOR_REQUEST`) and node reports to the shoreside community. The simulator continuously checks the connected vehicle's position against objects in the hazard field, and the sensor settings. When/if an object comes into sensor range, the simulator rolls the dice and if a detection is made, will send a `UHZ_DETECTION_REPORT` message to the vehicle. The vehicle may periodically re-configure its sensor setting by posting to `UHZ_CONFIG_REQUEST`. If the configuration request is acceptable, the simulator will respond with a message to `UHZ_CONFIG_ACK` bridged back out to the vehicle.

The hazard manager maintains a history of reported detections and listens for requests, via the variable `HAZARDSET_REQUEST`, to generate a *hazardset report*. It will respond to the request by posting a hazardset report in the variable `HAZARDSET_REPORT`.

26.2.1 Required MOOS Variable Bridges

Using `uFldHazardMgr` requires certain information flowing between the shoreside and vehicles communities as shown in Figure 99. The bridging is done by `pShare`, but the `pShare` configuration is handled dynamically using the `uFldNodeBroker` and `uFldShoreBroker` applications, we discuss here the necessary configuration entries for these two applications. From the vehicle to the shoreside, five variables need to be bridged. The below five lines should appear in the `uFldNodeBroker` configuration block on all vehicles.

```
// Bridges from Vehicle to Shoreside - in uFldNodeBroker configuration

bridge = src=APPCAST
bridge = src=NODE_REPORT_LOCAL, alias=NODE_REPORT
bridge = src=UHZ_SENSOR_CONFIG
bridge = src=UHZ_SENSOR_REQUEST
bridge = src=HAZARDSET_REPORT
```

The first two lines above would likely already be present due to their use in other applications. The latter three variables are generated by `uFldHazardMgr` with the intended recipient being `uFldHazardSensor` on the shoreside. The latter variable, `HAZARDSET_REPORT`, constitutes the hazardset report generated by `uFldHazardMgr`. The above five lines may also be found in the vehicle configuration for the Jake example mission discussed in Section 26.7.

The below four lines should appear in the `uFldShoreBroker` configuration block in the shoreside MOOS community. See Section 20 for a discussion on the syntax.

```
// Bridges from Shoreside to Vehicle - in uFldShore Broker configuration

bridge = src=APPCAST_REQ
bridge = src=UHZ_CONFIG_ACK_$V,           alias=UHZ_CONFIG_ACK
bridge = src=UHZ_DETECTION_REPORT_$V,     alias=UHZ_DETECTION_REPORT
bridge = src=HAZARDSET_REQUEST_$V,         alias=HAZARDSET_REQUEST
```

The first line deals with appcasting and would likely be present anyway due to its use in other applications as well. The second line allows sensor configuration acknowledgements to be sent to the vehicle (Section 26.2.4). The third line allows detection reports to be sent to the vehicle (Section 26.3). The last line allows an app running on the shoreside to bridge requests to the vehicle for a hazardset report. The above four lines may also be found in the shoreside configuration for the Jake example mission discussed in Section 26.7.

26.2.2 Configuration Parameters of `uFldHazardMgr`

The following parameters are defined for `uFldHazardMgr`. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so.

Listing 26.1: Configuration Parameters for `uFldHazardMgr`.

- `swath_width`: The desired sensor swath width. Legal values: the set of widths available to the sensor. The default is 25. If the requested swath width setting is not available, the result will be the closest setting available. [26.2.4](#).
- `pd`: The chosen probability of detection on the ROC curve determined by the sensor swath width. Legal values: the range [0, 1]. The default is 0.9. Section [26.2.5](#).

26.2.3 An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command-line:

```
$ uFldHazardMgr --example or -e
```

This will show the output shown in Listing 2 below.

Listing 26.2: Example configuration of the uFldHazardMgr application.

```

1 =====
2 uFldHazardMgr Example MOOS Configuration
3 =====
4
5 ProcessConfig = uFldHazardMgr
6 {
7     AppTick    = 4
8     CommsTick = 4
9
10    swath_width = 25          // the default
11    sensor_pd   = 0.9        // thd default
12 }
```

26.2.4 Configuring the Swath Width

Part of the initial responsibility of the hazard manager is to select the sensor settings. There are two primary settings, the `swath_width` and `sensor_pd` setting. The swath width refers to the width, stretching outward away from the sides of the vehicle. The selected width refers to the length on one side, so the *total* swath width is twice the `swath_width` setting.

Physical sensors are usually built with no more than a few swath width settings. These choices are known ahead of time. It is reasonable therefore to expect the configuration of `uFldHazardMgr` to reflect one of those choices. In our case, the available choices are the prevailing values configured of `uFldHazardSensor`. In the Jake example mission, the hazard sensor is configured to support sensor swath widths of 10, 25, and 50 meters. A sensor configuration request is made to the sensor by the hazard manager making a post to the `UHZ_CONFIG_REQUEST` variable similar to:

```
UHZ_CONFIG_REQUEST = "vname=archie,width=25,pd=0.85"
```

The request width sent by the hazard manager is set in the `uFldHazardMgr` configuration block: (See also Listing 2.)

```
swath_width = 25
```

The hazard sensor, once it has received the configuration request, posts a configuration acknowledgement, which is bridged back to the vehicle:

```
UHZ_CONFIG_ACK = "vname=archie,width=25,pd=0.85,pfa=0.53,pclass=0.91"
```

26.2.5 Configuring the Probability of Detection Setting

The second component of setting the sensor is choosing a probability of detection setting. The P_D is a number in the range of $[0, 1]$ and is accompanied by a corresponding probability of false alarm, P_{FA} . The relationship between P_D and P_{FA} is determined by the ROC curve related to the chosen sensor swath. This relationship is described in the documentation for `uFldHazardSensor` in Section 25.3.2, in Figure 92.

26.3 Under the Hood - Interacting with the Hazard Sensor

After the initial configuration, the hazard manager interacts with the hazard sensor by posting sensor requests, and periodically receiving detection reports. These may look something like:

```
UHZ_SENSOR_REQUEST = "vname=archie"
UHZ_DETECTION_REPORT = "x=-150.3,y=-117.5,label=12"
```

The request contains only the vehicle name. The hazard sensor is already receiving reports of the vehicle position, and has knowledge of the hazard field. So the sensor only needs to know that the vehicle indeed wishes to be sent detection reports.

Detection reports are only sent by the sensor when a detection is made. The simulated sensor does not send sensor images or data, but simulated *results* of sensor data, in the form of a declared detection. The simulated hazard sensor makes life a bit artificially easier by posting a detection *label*. This allows the user to make multiple passes over the same area and be sure that one hazard is not showing up as several hazards each with a slightly different location.

26.4 Under the Hood - Processing Data and Generating Reports

Hazardset reports are generated on-demand. First a request is received, immediately followed by the posting of a report. They may look similar to:

```
HAZARDSET_REQUEST = "true"
HAZARDSET_REPORT = "source=archie#x=-151,y=-217.3,label=01#x=-178.8,y=-234,label=15#
                     x=-59.8,y=-294.1,label=13#x=-150.3,y=-117.5,label=12#
                     x=-14.2,y=-293.60001,label=08#x=-65.8,y=-125.2,label=10"
```

The content of the `HAZARDSET_REQUEST` does not matter to `uFldHazardMgr`. It will interpret this mail as a request to generate a report regardless of the request content.

The `HAZARDSET_REPORT` content consists of a series of packets separated by the '#' character. The first packet names the source of the report and the remaining packets each declare the presence of a hazardous object and its location and label. The format of the report is embodied in the class `XYHazardSet` in `lib_ufld_hazards`. The class is populated with the hazards and the string representation is produced by calling the `getSpec()` function on a class instance.

As mentioned previously, `uFldHazardMgr` is a strawman approach for compiling sensor results and generating reports. The algorithm used here is dead-simple and easily improved upon: any detection ever made will be reported as a hazard. There is no further consideration of follow-on passes over the same area, even if the lack of detection may indicate that a prior initial detection was likely to be a false alarm.

26.5 Publications and Subscriptions for `uFldHazardMgr`

The interface for `uFldHazardMgr`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldHazardMgr --interface or -i
```

26.5.1 Variables Published by `uFldHazardMgr`

The primary output of `uFldHazardMgr` to the MOOSDB is the posting of requests for sensor information and the generation of hazardset reports.

- **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 26.6
- **HAZARDSET_REPORT**: A hazardset report summarizing the hazard manager's present belief about location of hazards. Section 26.4.
- **UHZ_CONFIG_REQUEST**: A message sent to the hazard simulator requesting a particular sensor configuration. Sections 26.2.4 and 26.2.5.
- **UHZ_SENSOR_REQUEST**: A message sent to `uFldHazardSensor` to request sensor/detection results be sent to the `uFldHazardMgr` as they become available. Section 26.3.

26.5.2 Variables Subscribed for by `uFldHazardMgr`

The `uFldHazardMgr` application will subscribe for the following four MOOS variables:

- **APPCAST_REQ**: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 11.10.4.
- **HAZARDSET_REQUEST**: A request asking `uFldHazardMgr` to produce immediately a hazardset report. Section 26.4.
- **UHZ_DETECTION_REPORT**: A report sent by the `uFldHazardSensor` indicating the detection of a hazardous object and its location. Section 26.3.

- **UHZ_CONFIG_ACK**: A message sent by **uFldHazardSensor** confirming the requested sensor configuration information. Sections 26.2.4 and 26.2.5.

Command Line Usage of uFldHazardMgr

The **uFldHazardMgr** 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:

```
$ uFldHazardMgr --help or -h
```

This will show the output shown in Listing 3 below.

Listing 26.3: Command line usage for uFldHazardMgr.

```

1 =====
2 Usage: uFldHazardMgr file.moos [OPTIONS]
3 =====
4
5 Options:
6   --alias=<ProcessName>
7     Launch uFldHazardMgr with the given process name.
8   --example, -e
9     Display example MOOS configuration block.
10  --help, -h
11    Display this help message.
12  --interface, -i
13    Display MOOS publications and subscriptions.
14  --version,-v
15    Display release version of uFldHazardMgr.

```

26.6 Terminal and AppCast Output

The **uFldHazardMgr** application produces some useful information to the terminal and identical content through appcasting. An example is shown in Listing 4 below. On line 2, the name of the local community or vehicle name is listed on the left. On the right, "0/0(1414) indicates there are no configuration or run warnings, and the current iteration of **uFldHazardMgr** is 1414. Lines 4-13 convey the requested and prevailing sensor configuration settings.

Listing 26.4: Example uFldHazardMgr console output.

```

1 =====
2 uFldHazardMgr archie                      0/0(1414)
3 =====
4 Config Requested:
5   swath_width_desired: 38
6           pd_desired: 0.86
7 config requests sent: 118
8           acked: 1

```

```

9 -----
10 Config Result:
11     config confirmed: true
12     swath_width_granted: 50
13         pd_granted: 0.86
14
15 -----
16
17     sensor requests: 2003
18     detection reports: 7
19
20 Hazardset Reports Requested: 1
21     Hazardset Reports Posted: 1
22
23 =====
24 Most Recent Events (7):
25 =====
26 [1046.43]: New Detection, label=08, x=-14.2, y=-293.6
27 [935.39]: New Detection, label=08, x=-14.2, y=-293.6
28 [928.03]: New Detection, label=13, x=-59.8, y=-294.1
29 [799.39]: New Detection, label=12, x=-150.3, y=-117.5
30 [700.65]: New Detection, label=13, x=-59.8, y=-294.1
31 1[525.17]: New Detection, label=15, x=-178.8, y=-234.0
32 [522.15]: New Detection, label=01, x=-151.0, y=-217.3

```

Lines 17 shows the number of sensor requests sent to the hazard sensor. This message is sent continuously so it is not surprising to be high. Line 18 shows the number of detection reports received. Lines 20-21 show the number of time a hazardset report has been requested and posted. The events in lines 23-32 are typically only contain the arrival of a new detections from the sensor simulator.

26.7 The Jake Example Mission Using uFldHazardMgr

The *Jake* mission is distributed with the MOOS-IvP source code and contains a ready example of the `uFldHazardMgr` application, configured with hazard field in an included text file. Assuming the reader has downloaded the source code available at www.moos-ivp.org and built the code according to the discussion in Section 1.2.9, the *Jake* mission may be launched by:

```
$ cd moos-ivp/ivp/missions/m10_jake/
$ ./launch.sh 10
```

The argument, 10, in the line above will launch the simulation in 10x real time. Once this launches, the `pMarineViewer` GUI application should launch and the mission may be initiated by hitting the DEPLOY button.

27 uFldHazardMetric: Grading a HazardSet Report

27.1 Overview

The `uFldHazardMetric` application is a utility for quickly evaluating a hazardset report; a list of declared hazards and their locations. Evaluating a hazardset report against ground truth and a reward structure is fairly straight-forward, but tedious. This tool performs this operation automatically, and as a MOOS process with the result posted both to the MOOSDB and viewable in the appcast output of `uFldHazardMetric`. Operation is comprised of a few simple parts:

1. *Import a ground-truth hazard field:* A ground truth hazard field is a text file listing the location of hazards and hazard-like objects, and their locations. This file also typically includes a search region, a convex polygon containing all listed objects. A `uFldHazardMetric` configuration parameter names the file.
2. *Import a reward structure:* A reward structure, consisting of penalties for missed hazards and false alarms, is imported as a set of `uFldHazardMetric` configuration parameter.
3. *Evaluate a hazardset report:* A hazardset report is received by MOOS mail and evaluated item by item against the ground truth and reward structure. The results are then posted and rendered. This step is repeated for each received report.

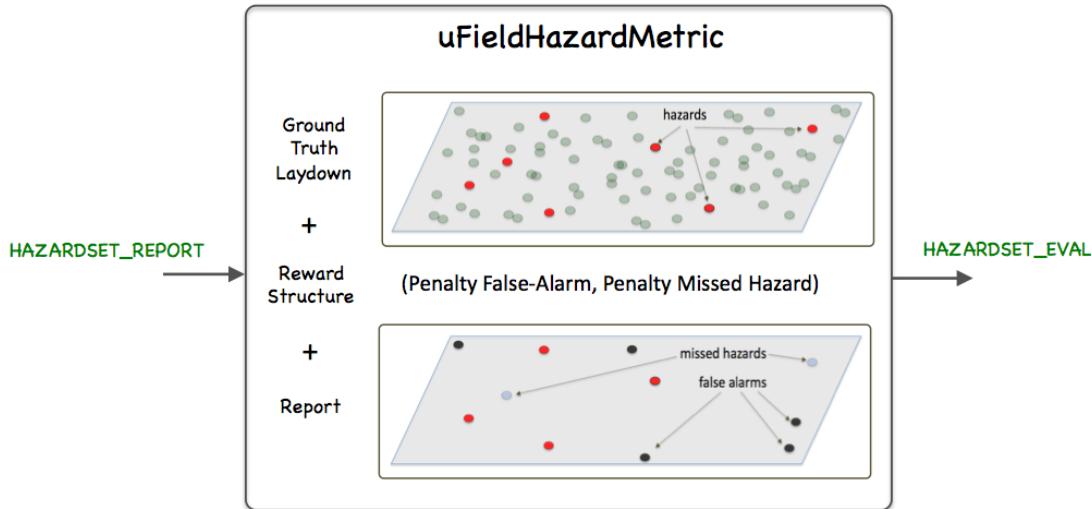


Figure 100: **The `uFldHazardMetric`:** interacts with the on-board sensor and processes sensor information and generates a report, upon request, regarding the identification and location of hazards. The arrows indicate the key MOOS variables used for interacting with the sensor and generating reports.

27.2 Using `uFldHazardMetric`

Typical use of `uFldHazardMetric` has it situated in the shoreside community, with hazardset reports shared from vehicles to the shoreside, and evaluation results shown via the `uFldHazardMetric` appcast

output running on `pMarineViewer`. Full evaluation reports are also logged to the shoreside log file for later reference. This usage scenario with variations is described next. An example of this usage is in the Jake example mission described in Section ??.

Typical Module Topology

The typical module topology is shown in Figure 101 below. The `uFldHazardMetric` is situated in the shoreside MOOS community. It does *not* interact with the `uFldHazardSensor` directly, but they are typically both configured with the same ground truth hazard file. `HAZARDSET_REPORT` messages are assumed to come from the vehicle. In the usage case below, they are produced by `uFldHazardMgr`. But as the latter is simply a strawman sensor processing module, that could be replaced with something else entirely. `HAZARDSET_REPORT_EVAL` messages may be shared back to the vehicle, but this is likely not essential, as the typical destination of an evaluation is the appcast output and the log file.

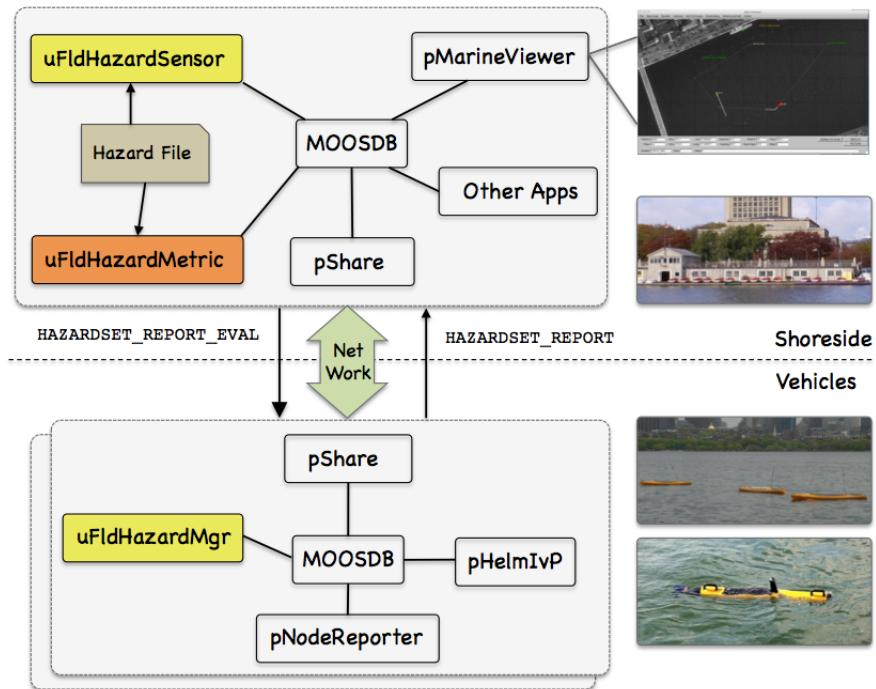


Figure 101: **Typical `uFldHazardMetric` Topology:** This module runs on the shoreside, alongside the hazard sensor typically, and receives hazardset reports from the vehicle bridge with the `HAZARDSET_REPORT` variable. Evaluations may be seen via the appcast output of `uFldHazardMetric`, or in the log file.

27.2.1 Required MOOS Variable Bridges

Using `uFldHazardMetric` requires certain information flowing between the shoreside and vehicle communities as shown in Figure 101. Bridging is done by `pShare`, but the `pShare` configuration is handled dynamically using the `uFldNodeBroker` and `uFldShoreBroker` applications. We discuss here the necessary configuration entries for these two applications. From the vehicle to the shoreside,

one variable needs to be shared. The below line should appear in the `uFldNodeBroker` configuration block on all vehicles.

```
bridge = src=HAZARDSET_REPORT      // in uFldNodeBroker config block
```

This `HAZARDSET_REPORT` variable constitutes the report generated by `uFldHazardMetric` or a similar module running in the vehicle generating a hazardset report. The line may also be found in the vehicle configuration for the Jake example mission discussed in Section 27.6.

Going in the other direction, from shoreside to vehicle, the below line should appear in the `uFldShoreBroker` configuration block in the shoreside MOOS community. See Section 20 for a discussion on the syntax.

```
// Bridge from Shoreside to Vehicle - in uFldShore Broker configuration
bridge = src=HAZARDSET_REPORT_EVAL_$V, alias HAZARDSET_REPORT_EVAL
```

It may not be the case that your vehicle is actually utilizing the report evaluation, so the above may be optional. And certainly a typical mission will need to share other variables besides these, but from the perspective of `uFldHazardMetric`, these are the shares to make sure are configured. The above couple lines may also be found in the shoreside configuration for the Jake example mission discussed in Section 27.6.

27.2.2 The False-Alarm and Missed-Hazard Reward Structure

The primary metric for evaluating a hazardset report is based on penalties assigned to missed hazards and false alarms. The penalties are set with the parameters:

```
penalty_missed_hazard = <number>    // The default is 100
penalty_false_alarm   = <number>    // The default is 10
```

With k_1 missed hazards and k_2 false alarms, the penalty is:

$$\text{penalty}(k_1, k_2) = \text{penalty}_{MH}(k_1) + \text{penalty}_{FA}(k_2)$$

27.2.3 The Max-Time and Time-Overage Reward Structure

An optional additional metric may be applied which penalizes the report if it is late, with additional potential penalties the longer it is late. The time penalties are set with following parameters, beginning with the `max_time` parameter setting the point when a report is considered *late*:

```
max_time          = <number> // seconds, default is 0
penalty_max_time_over = <number> // penalty units, default is 0
penalty_max_time_rate = <number> // penalty units, default is 0
```

The `penalty_max_time_over` parameter indicates the immediate one-time penalty applied if the report is late at all. The `penalty_max_time_overage` penalty is applied for *each second* of time past the deadline. If `max_time` is zero, there is no mission time limit.

If t is the amount of time over the max time:

$$\text{penalty}(k_1, k_2, t) = \begin{cases} \text{penalty}_{FA}(k_1) + \text{penalty}_{MH}(k_2) & t \leq 0 \\ \text{penalty}_{FA}(k_1) + \text{penalty}_{MH}(k_2) + \text{penalty}_{TO} + \text{penalty}_{TR}(t) & t > 0 \end{cases}$$

The search duration clock re-starts each time `uFldHazardMetric` receives incoming mail on the variable `HAZARD_SEARCH_START`, regardless of the variable's value. Typically this variable is posted upon vehicle deployment as is done in the Jake example mission. (Hint: see how `button_one` is configured for `pMarineViewer` in the Jake mission.)

27.2.4 Raw and Normalized Scores

Past experience has shown that people appreciate a normalized score. A goal of zero (no penalties, perfect score) is somehow not as motivating as striving for 100% on a scale of zero to 100. A normalized score is derived from considering the worst possible score if each object in the hazard file where reported wrong. The score may be worse than this if the report is late and there are late penalties, but time not used for the purposes of normalizing.

If j_1 and j_2 are the actual number of hazards and benign objects taken from ground truth in the hazard file, the worst score (without applying overtime penalties) is:

$$\text{maxpenalty}(j_1, j_2) = \text{penalty}_{MH}(j_1) + \text{penalty}_{FA}(j_2)$$

The normalized score is then:

$$\text{score}(k_1, k_2, t) = \frac{\text{maxpenalty}(j_1, j_2) - \text{penalty}(k_1, k_2, t)}{\text{maxpenalty}(j_1, j_2)}$$

If $\text{penalty}(k_1, k_2, t)$ is actually greater than $\text{maxpenalty}(j_1, j_2)$ due to lateness, resulting in a negative score, the normalized score is clipped to zero.

27.2.5 The Report Evaluation Format

The evaluation of the hazardset report has two formats, a terse and and verbose form. The terse form, `HAZARDSET_EVAL`, fully explains the score, the metrics, and the components of the submitted report responsible for the score. It may looks something like the example below from the Jake example mission:

```
HAZARDSET_EVAL = vname=archie, report_name=BillandJoe,
                  total_score=675, norm_score=37.5,
                  score_missed_hazards=500, score_false_alarms=175,
                  score_time_overage=0, total_objects=10,
                  total_time=1284.91, received_time=1314.05,
                  start_time=29.14, missed_hazards=5,
                  correct_hazards=5, false_alarms=5,
                  penalty_false_alarm=35, penalty_missed_hazard=100,
                  penalty_max_time_over=100, penalty_max_time_rate=0.05,
                  max_time=1800
```

The full evaluation, `HAZARDSET_EVAL_FULL` provides all the details about which hazards were declared and missed, and which benign objects were false alarms. It may look something like the example below from the Jake example mission:

```
HAZARDSET_EVAL_FULL = (Everything in the normal report),object_report={  
    label=01,truth=hazard,report=hazard#  
    label=02,truth=hazard,report=nothing,penalty=100#  
    label=03,truth=hazard,report=hazard#  
    label=04,truth=hazard,report=nothing,penalty=100#  
    ...  
    label=15,truth=benign,report=hazard,penalty=35#  
    label=16,truth=benign,report=hazard,penalty=35#  
    label=17,truth=benign,report=nothing#  
    label=18,truth=benign,report=hazard,penalty=35}
```

The latter may only be used for forensics, or perhaps if further clarity is needed in how a scoring was applied.

27.3 Configuration Parameters of uFldHazardMetric

The following parameters are defined for `uFldHazardMetric`. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so.

Listing 27.1: Configuration Parameters for `uFldHazardMetric`.

- `penalty_missed_hazard`: The penalty for a missed hazard. The default is 100. Section [27.2.2](#).
- `penalty_false_alarm`: The penalty for a false alarm. The default is 10. Section [27.2.2](#).
- `penalty_max_time_over`: The penalty for submitting a report late. The default is zero. Section [27.2.3](#).
- `penalty_max_time_verage`: The penalty for submitting a late report, applied to every second it is late. The default is zero. Section [27.2.3](#).
- `max_time`: The time after which a submitted report is considered late. The default is zero, indicating there is no time limit. Section [27.2.2](#).
- `hazard_file`: The name of a hazard file naming the ground truth hazard field.

An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command-line:

```
$ uFldHazardMetric --example or -e
```

This will show the output shown in Listing 2 below.

Listing 27.2: Example configuration of the `uFldHazardMetric` application.

```

1 =====
2 uFldHazardMetric Example MOOS Configuration
3 =====
4
5 ProcessConfig = uFldHazardMetric
6 {
7     AppTick      = 4
8     CommsTick   = 4
9
10    penalty_missed_hazard = 100    // default
11    penalty_false_alarm   = 10    // default
12    penalty_max_time_over = 0     // default
13    penalty_max_time_rate = 0     // default
14
15    max_time        = 0          // default (no time limit)
16    hazard_file     = hazards.txt
17 }

```

27.4 Publications and Subscriptions for uFldHazardMetric

The interface for `uFldHazardMetric`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldHazardMetric --interface or -i
```

27.4.1 Variables Published by uFldHazardMetric

- **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 27.5
- **HAZARDSET_EVAL**: The shorter version of a hazardset report evaluation. Section 27.2.5.
- **HAZARDSET_EVAL_FULL**: The longer version of a hazardset report evaluation. Section 27.2.5.
- **HAZARDSET_EVAL_<VNAME>**: The shorter version of a hazardset report evaluation. The vehicle from which the report was received is appended to the evaluation so it may be bridged only back to that vehicle. Section 27.2.5.
- **HAZARDSET_EVAL_<VNAME>**: The longer version of a hazardset report evaluation. The vehicle from which the report was received is appended to the evaluation so it may be bridged only back to that vehicle. Section 27.2.5.
- **HAZARD_SEARCH_SCORE**: The normalized score reported in HAZARDSET_EVAL published as a single numerical value.

27.4.2 Variables Subscribed for by uFldHazardMetric

The `uFldHazardMetric` application will subscribe for the following four MOOS variables:

- **APPCAST_REQ**: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 11.10.4.

- **HAZARDSET_REPORT**: An incoming hazardset report. Section 26.4.
- **HAZARD_SEARCH_START**: An indication that the clock used to apply time limits and penalties is to be restarted. Section 27.2.3.

Command Line Usage of uFldHazardMetric

The **uFldHazardMetric** 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:

```
$ uFldHazardMetric --help or -h
```

This will show the output shown in Listing 3 below.

Listing 27.3: Command line usage for uFldHazardMetric.

```

1 =====
2 Usage: uFldHazardMetric file.moos [OPTIONS]
3 =====
4
5 Options:
6   --alias=<ProcessName>
7     Launch uFldHazardMetric with the given process name.
8   --example, -e
9     Display example MOOS configuration block.
10  --help, -h
11    Display this help message.
12  --interface, -i
13    Display MOOS publications and subscriptions.
14  --version,-v
15    Display release version of uFldHazardMetric.

```

27.5 Terminal and AppCast Output

The **uFldHazardMetric** application produces some useful information to the terminal and identical content through appcasting. An example is shown in Listing 4 below. On line 2, the name of the local community, typically the shoreside community, is listed on the left. On the right, "0/0(5429) indicates there are no configuration or run warnings, and the current iteration of **uFldHazardMetric** is 5429. Lines 4-6 show the name of the ground truth hazard file and the number of hazards and benign objects. Lines 8-13 convey the requested and prevailing configuration settings for evaluating incoming reports.

Listing 27.4: Example uFldHazardMetric console output.

```

1 =====
2 uFldHazardMetric shoreside          0/0(5429)
3 =====
4 Hazard File: (hazards.txt)

```

```

5      Hazard: 10
6      Benign: 8
7
8 Reward Structure:
9      Penalty Missed Hazard: 100
10     Penalty False Alarm: 35
11     Penalty Max Time Over: 100
12     Penalty Max Time Rate: 0.05
13             Max Time: 1800
14
15 =====
16 Received Reports: 6
17 Elapsed Time: 1285.76
18 =====
19 Report   Total    Time     Time     Raw      Norm
20 Name     Reports  Received  Elapsed  Score    Score
21 -----  -----
22 Sarah    6        1351.97  1278.28 675     37.5
23
24 =====
25 Most Recent Report: (archie/Sarah)
26         total_score: 675 (37.5)
27     score_missed_hazards: 500 (5)
28     score_false_alarms: 175 (5)
29     score_time_overage: 0 (0)
30 -----
31     objects reported: 10
32     correct_hazards: 5 (of 10)
33
34 =====
35 Most Recent Events (7):
36 =====
37 [1351.97]: Received valid report from: archie
38 [1347.46]: Received valid report from: archie
39 [1181.93]: Received valid report from: archie
40 [1178.42]: Received valid report from: archie
41 [641.28]: Received valid report from: archie
42 [631.76]: Received valid report from: archie
43 [0.00]: Reading hazards.txt: Objects read: 18

```

Lines 15-22 provide a summary of reports received so far, perhaps from multiple vehicles. Line 16 shows the number of received reports total from *all* vehicles, and line 17 shows the elapsed time since the receipt of `HAZARD_SEARCH_START` as discussed in Section 27.2.3. Beginning with line 19, for each vehicle the total reports received are shown, plus information about the last received report from that vehicle. The last four columns show (a) the time the report was received, (b) the elapsed time for the report since the latest timer reset, (c) the raw score as discussed in Sections 27.2.2 and 27.2.3, and (d) the normalized score as discussed in Section 27.2.4.

Lines 24-32 dive into more detail about the latest report received from any vehicle. Line 25 shows the name of the report which may consist of both a vehicle name and additional report name. The total raw and normalized score is shown next on line 26, with the justification for the score shown next in lines 27-32. Following this block of output, events are shown starting here on line 34. In this case most events simply report the arrival of new reports, but other events may indicate anomalous activities such as the arrival of an empty report.

27.6 The Jake Example Mission Using uFldHazardMetric

The *Jake* mission is distributed with the MOOS-IvP source code and contains a ready example of the `uFldHazardMetric` application, configured with a hazard field in an included text file. Assuming the reader has downloaded the source code available at www.moos-ivp.org and built the code according to the discussion in Section 1.2.9, the *Jake* mission may be launched by:

```
$ cd moos-ivp/ivp/missions/m10_jake/
$ ./launch.sh 12
```

The argument, 12, in the line above will launch the simulation in 12x real time. Once this launches, the `pMarineViewer` GUI application should launch and the mission may be initiated by hitting the `DEPLOY_ALL` button. Shortly thereafter, two vehicles named `archie` and `betty` will begin simple lawnmower patterns over half of the search region each, as shown in Figure 102 below.

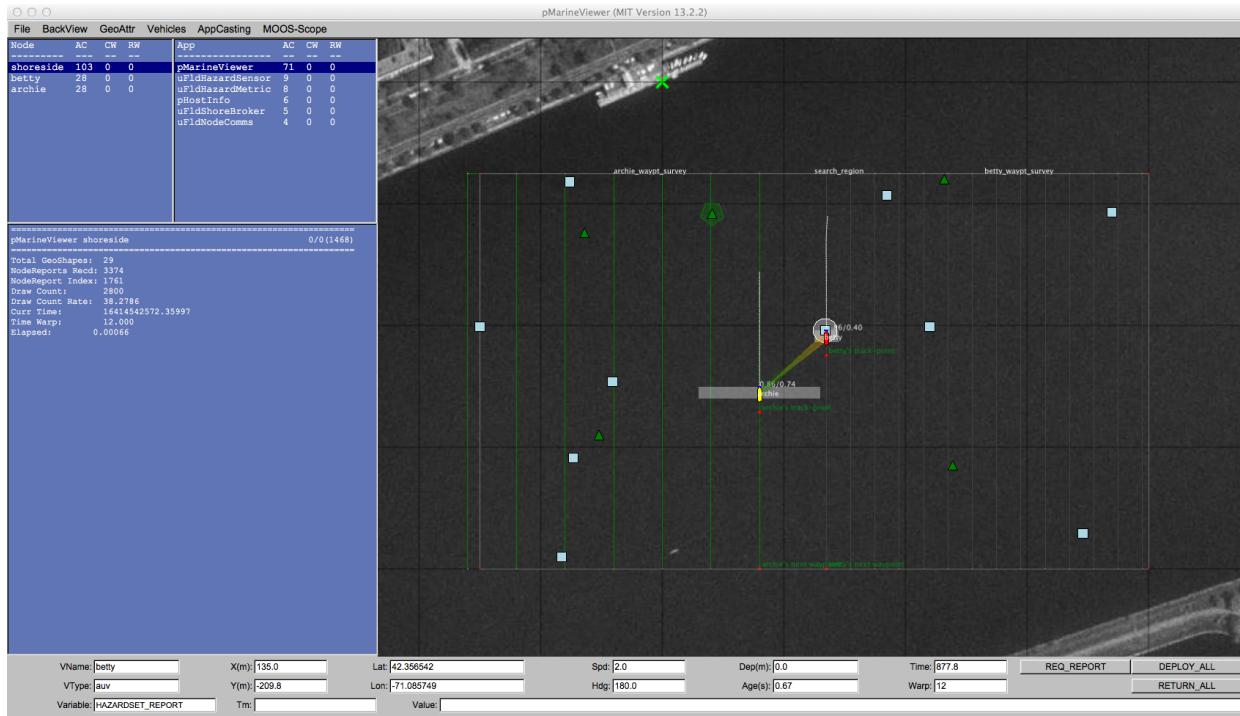


Figure 102: **uFldHazardMetric in the Jake example mission:** The output of `uFldHazardMetric` is shown in the appcast panel in the lower left after a hazardset report has been received and evaluated.

Any time after the vehicle has been deployed, the user may request the generation of a hazardset report. The `REQ_REPORT` button sends a `HAZARDSET_REQUEST` message to the vehicles, each running `uFldHazardMgr`. Repeated requests result in updated reports. The overall score of the reports tends higher as the mission progresses and more hazards are detected.

28 The Jake Example Mission

The *Jake* mission is distributed with the MOOS-IvP code and was designed to highlight four things:

- The `uFldHazardSensor` application,
- The `uFldHazardMetric` application,
- The `uFldHazardMgr` application, and
- How each of the above work together, in a straw-man solution to the hazard search challenge problem.

The example mission contains a default hazard field and a few other hazard fields for testing. Assuming the reader has downloaded the source code available at www.moos-ivp.org and built the code according to the discussion in Section ??, the *Jake* mission may be launched by:

```
$ cd moos-ivp/ivp/missions/m10_jake/
$ ./launch.sh 12
```

The argument, 12, in the line above will launch the simulation in 12x real time. Once this launches, the `pMarineViewer` GUI application should launch and the mission may be initiated by hitting the `DEPLOY_ALL` button. Shortly thereafter, two vehicles named `archie` and `betty` will begin simple lawnmower patterns over half of the search region each, as shown in Figure 103 below.

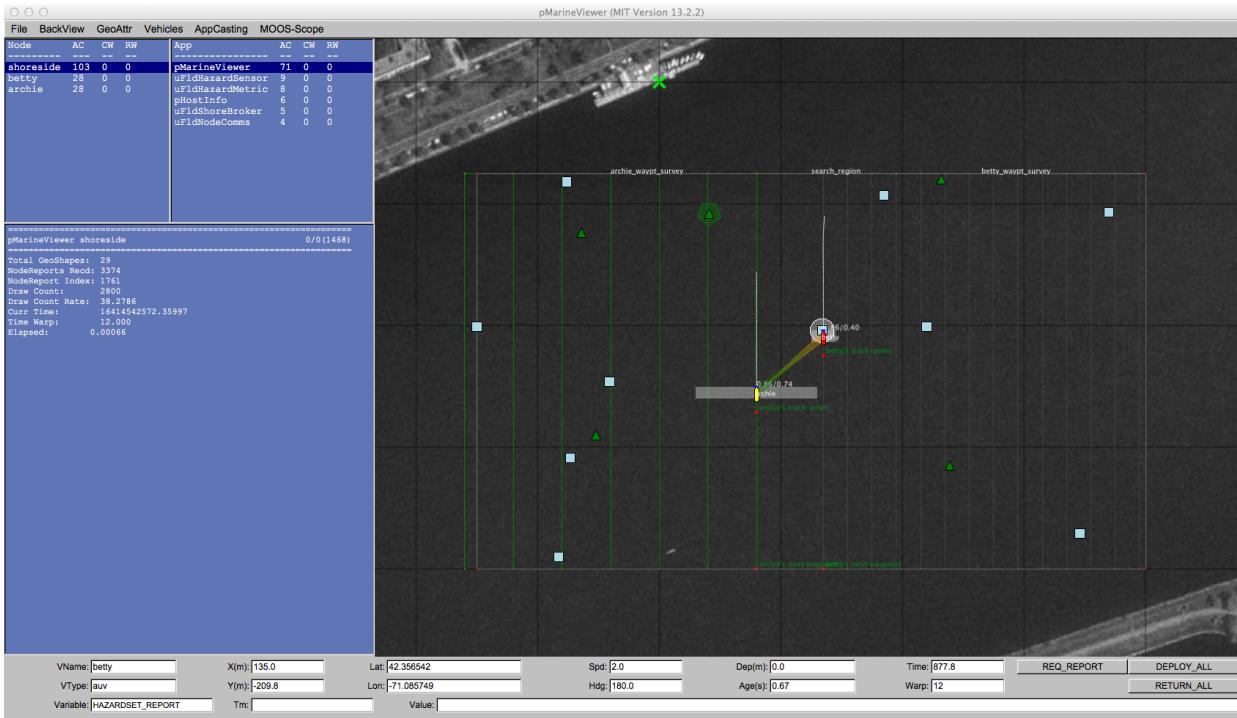


Figure 103: **The Jake example mission:** Two vehicles are on independent lawn-mower shaped search patterns, each with a chosen swath width. Each vehicle will generate a hazardset report upon completion of their initial search. This is a straw-man solution to the hazard-search problem. Several changes or additions need to be made to improve the performance, but this mission constitutes a syntactically valid hazard-search mission.

Any time after the vehicle has been deployed, the user may request the generation of a hazardset report. The `REQ_REPORT` button sends a `HAZARDSET_REQUEST` message to the vehicles, each running `uFldHazardMgr`. Repeated requests result in updated reports. The overall score of the reports tends higher as the mission progresses and more hazards are detected.

28.1 Module Topology in the Jake Mission

The key components of the Jake mission are shown in Figure 104 below. The `uFldHazardSensor` and `uFldHazardMetric` apps run on the shoreside, the `uFldHazardMetric` runs on the vehicle. The latter is a straw-man implementation of a application to be implemented by individual developers in the hazard search competition.

All three applications are described in detail in their own separate sections (Sections 25, 27, and 26). A high-level description of operation is that the vehicles (a) configure the hazard sensor, (b) process the sensor information as they move through the field, and finally (c) generate a hazardset report and send it back to the shoreside for evaluation.

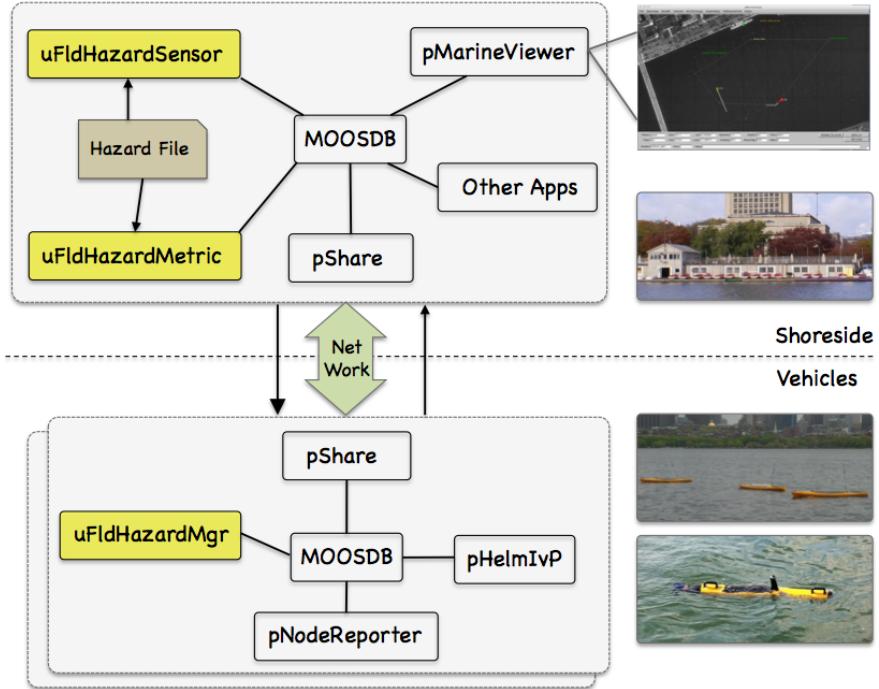


Figure 104: **The Jake Example Mission Module Topology:** The three key modules of the Jake mission are `uFldHazardSensor`, `uFldHazardMetric`, and `uFldHazardMgr`.

28.2 Key MOOS Variables in the Jake Mission

To digest what is going on in this mission, familiarity with the key MOOS variables published by each key modules is needed. We list these here for reference in the following discussion operation. It's worth remembering that this information is always available on the command-line, by typing for example:

```
$ uFldHazardSensor --interface or -i
```

The above also provides example values for the key variables, whereas below just the variable names are listed.

uFldHazardSensor Interface

- Publications: `UHZ_DETECTION_REPORT`, `UHZ_HAZARD_REPORT`, `UHZ_CONFIG_ACK`, `UHZ_OPTIONS_SUMMARY`, `VIEW_CIRCLE`, `VIEW_MARKER`, `VIEW_POLYGON`.
- Subscriptions: `NODE_REPORT`, `UHZ_SENSOR_REQUEST`, `UHZ_CONFIG_REQUEST`, `UHZ_CLASSIFY_REQUEST`.

uFldHazardMetric Interface

- Publications: `UHZ_MISSION_PARAMS`, `HAZARDSET_EVAL`.
- Subscriptions: `HAZARD_SEARCH_START`, `HAZARDSET_REPORT`.

uFldHazardMgr Interface

This information, and example values for each variable, may be obtained from the command-line by typing `uFldHazardMgr -i`.

- Publications: `UHZ_CONFIG_REQUEST`, `UHZ_SENSOR_REQUEST`, `HAZARDSET_REPORT`.
- Subscriptions: `HAZARDSET_REQUEST`, `UHZ_CONFIG_ACK`, `UHZ_DETECTION_REPORT`, `UHZ_OPTIONS_SUMMARY`.

28.3 General Description of Events in the Jake Mission

There are roughly three phases to the Jake example mission, (a) initialization and startup, (b) search, and (c) reporting results. These are described below with some reference to what additional measures would be needed beyond the straw-man mission.

28.3.1 Phase 1: Sensor Configuration and Handling of Mission Parameters

For each vehicle, there are two steps initialization phase, (a) initializing the hazard sensor, and (b) digesting the mission parameters.

Sensor Configuration

The hazard sensor initialization is done in three steps. First `uFldHazardSensor` publishes the sensor options, with something similar to:

```
UHZ_OPTIONS_SUMMARY = width=10,exp=6,class=0.91:width=25,exp=4,class=0.85:  
width=50,exp=2,class=0.78
```

Once the vehicle knows the sensor options, it can pick one. In this case the `uFldHazardMgr`, running on vehicle `archie` posts something similar to:

```
UHZ_CONFIG_REQUEST = vname=archie,width=25,pd=0.9
```

Once the request has been received, `uFldHazardSensor`, will either accept the requested swath width outright, or map it to the closest legal option. After setting the value for P_D , it will determine the P_{FA} and P_C , and post a confirmation to the following variable:

```
UHZ_CONFIG_ACK = vname=archie,width=20,pd=0.9,pfa=0.53,pclass=0.91
```

Once this is done, the sensor is configured. In the Jake mission, being a straw-man mission, some of the above steps are over-simplified. In the Jake mission the vehicle does not handle the options summary and just blindly requests a particular sensor setting. It also does not handle the configuration acknowledgement. This may be something improved by a user extending this mission.

Handling of Mission Parameters

Upon startup, the mission parameters are published by `uFldHazardMetric` on the shoreside and sent to each of the vehicles. The format of this message may look something like:

```
UHZ_MISSION_PARAMS = penalty_missed_hazard=100,  
                      penalty_false_alarm=35,  
                      penalty_max_time_over=200,  
                      penalty_max_time_rate=0.45,  
                      search_region = pts={-150,-75:-150,-50:40,-50:40,-75}
```

The straw-man Jake mission however does not consume and react to these mission parameters. The search mission pattern is hard-coded and the penalties are disregarded, all detections are reported as hazards. This will almost certainly need to be handled by users wishing to improve on the baseline performance provided in the Jake mission.

28.3.2 Phase 2: Executing the Search Phase

The activities during the search phase consist of two types of messages originating in the vehicle, and two types of messages originating in the hazard sensor back to the vehicle. Basic sensor operation is turned on when the vehicle publishes the variable:

```
UHZ_SENSOR_REQUEST = vname=archie
```

In the Jake mission, the above message originates in `uFldHazardMgr`. This is received by the `uFldHazardSensor` on the shoreside. The hazard sensor regards the sensor to be on if, for the named vehicle, it has received a sensor request recently. If the sensor is on and the vehicle passes over an object and produces a detection, then a detection report is posted by the sensor and sent back to the vehicle:

```
UHZ_DETECTION_REPORT = x=51,y=11.3,label=12
```

The vehicle may decide to request a classification report for a given detection and publish:

```
UHZ_CLASSIFY_REQUEST = vname=archie,label=12,priority=80
```

However, in the Jake mission, no classify requests are generated. The vehicle simply interprets all detections to be hazards and generates a report without using the classify capability of the hazard sensor. This is another area where the user wishing to improve on the baseline mission would focus their effort. Classify requests are eventually answered by the `uFldHazardSensor` publishing:

```
UHZ_HAZARD_REPORT = x=51,y=11.3,type=hazard,label=12
```

28.3.3 Phase 3: Reporting Results

The final phase of the mission involves sending a report from the vehicle to the shoreside, handled by `uFldHazardMetric`:

```
HAZARDSET_REPORT = source=archie#
    x=-151,y=-217.3,label=01#
    x=-178.8,y=-234,label=15#
    x=-59.8,y=-294.1,label=13
```

When is this report sent? In the Jake mission, this report is sent simply after the vehicles have completed their lawnmower pattern. Furthermore, in the Jake mission, each vehicle sends their own report; there is no coordination or communication. The `uFldHazardMetric` evaluates a *single* latest report, so the earlier report in the Jake mission is simply dropped. This is another glaring issue for improvement.

After receiving and evaluating a report, the `uFldHazardMgr` publishes the evaluation to the variable, consisting partly of results, and partly reiterating the criteria for evaluation:

```
HAZARDSET_EVAL = vname=archie,
    report_name=archie_team,
    total_score=675,
    norm_score=37.5,
    score_missed_hazards=500,
    score_false_alarms=175,
    score_time_overage=0,
    total_objects=10,
    total_time=1284.91,
    received_time=1314.05,
    start_time=29.14,
    missed_hazards=5,
    correct_hazards=5,
    false_alarms=5,
    penalty_false_alarm=35,
    penalty_missed_hazard=100,
    penalty_max_time_over=100,
    penalty_max_time_rate=0.05,
    max_time=1800
```

In the Jake mission, the `uFldHazardMgr` also will publish a hazardset report upon request when it receives incoming mail `HAZARDSET_REQUEST=true`. In the Jake mission, the `pMarineViewer` GUI is configured to generate this request with one of the custom action buttons.

28.4 Required MOOS Variable Bridges

The Jake mission requires MOOS variables to be shared between the vehicle and shoreside community depicted in Figure 104. The variables of key processes were described above in Sections 28.2 and 28.3. The sharing (also referred to as bridging) between MOOS communities is done with the `pShare` MOOS app distributed with MOOS. Configuration involves specifying which variables

are to be sent to which other machine (IP address) along with the port where the other [MOOSDB](#) resides. This is configured on the shoreside for variables shared *from* the shoreside *to* the vehicles. A similar configuration exists on the vehicles for variables going in the other direction.

28.4.1 Variable Bridges from the Shoreside to the Vehicle

Many of the variables shared from the shoreside to the vehicle were discussed in the previous sections, [28.2](#) and [28.3](#). The variable share configuration is handled on the shoreside with the [uFldShoreBroker](#) application. Below is the configuration used in the Jake mission:

```
ProcessConfig = uFldShoreBroker
{
    AppTick      = 1
    CommsTick    = 1

    // Note: [qbridge = FOO]  is shorthand for
    //        [bridge = src=FOO_$V, alias=FOO] and
    //        [bridge = src=FOO_ALL, alias=FOO]

    qbridge = DEPLOY, RETURN, NODE_REPORT, NODE_MESSAGE
    qbridge = MOOS_MANUAL_OVERRIDE

    bridge   = src=APPCAST_REQ
    bridge   = src=UHZ_MISSION_PARAMS
    bridge   = src=UHZ_OPTIONS_SUMMARY

    bridge   = src=UHZ_CONFIG_ACK_$V,           alias=UHZ_CONFIG_ACK
    bridge   = src=UHZ_HAZARD_REPORT_$V,         alias=UHZ_HAZARD_REPORT
    bridge   = src=UHZ_DETECTION_REPORT_$V,       alias=UHZ_DETECTION_REPORT
    bridge   = src=HAZARDSET_REQUEST_ALL,         alias=HAZARDSET_REQUEST
}
```

In addition to the sensor variables discussed previously, certain basic mission variables, e.g., [DEPLOY](#), [RETURN](#) are shared to allow for basic command-and-control of the vehicles. More on [uFldShoreBroker](#) application can be found in the documentation for that application. In short, the application allows for configurable autonomic point-to-point sharing of information as vehicles become known.

28.4.2 Variable Bridges from the Vehicle to the Shoreside

Most of the variables shared from the vehicle to the shoreside were discussed in the previous sections, [28.2](#) and [28.3](#). The variable share configuration, for variables sent from the vehicle to the shoreside, is handled on vehicle with the [uFldNodeBroker](#) application. Below is the configuration used in the Jake mission:

```
ProcessConfig = uFldNodeBroker
{
    AppTick      = 1
    CommsTick    = 1
```

```

TRY_SHORE_HOST = pshare_route=multicast_9

BRIDGE = src=VIEW_POLYGON
BRIDGE = src=VIEW_POINT
BRIDGE = src=VIEW_SEGLIST
BRIDGE = src=APPCAST
BRIDGE = src=UHZ_CLASSIFY_REQUEST
BRIDGE = src=UHZ_SENSOR_REQUEST
BRIDGE = src=UHZ_CONFIG_REQUEST
BRIDGE = src=HAZARDSET_REPORT
BRIDGE = src=NODE_REPORT_LOCAL, alias=NODE_REPORT
BRIDGE = src=NODE_MESSAGE_LOCAL, alias=NODE_MESSAGE
}

```

In addition to the sensor variables discussed previously, a few geometry variables, e.g., `POLYGON`, are shared to the shore for rendering things such as the vehicle search path or search region in the shoreside display. More on the `uFltNodeBroker` application can be found in the documentation for that application. In short, the application allows for configurable automatic point-to-point sharing of information as the location of the shoreside is discovered, without requiring changes to the mission files if the shoreside location, i.e., IP address, changes.

28.5 Hazard Files in the Jake Mission

There are several example hazard files included in the Jake mission. The default file used is `hazards.txt`, but there are additionally four other files:

- `hazards_01.txt`
- `hazards_02.txt`
- `hazards_03.txt`
- `hazards_04.txt`

They vary in (a) number of benign objects, (b) number of hazards, (c) hazard resemblance of the benign objects, and (d) aspect sensitivity of objects. Each of these files was created with the `gen_hazards` utility distributed with moos-ivp. The first line of each file contains a comment, a line showing exactly the command-line invocation of `gen_hazards` used in making the file. You are welcome to create your own to test situations beyond those in the handful of files included in the Jake mission.

To launch the mission with a different hazard file, the `--hazards` command-line switch is supported with the provided launch script. For example:

```
$ ./launch.sh --hazards=hazards_03.txt 10
```

As before, the 10 on the command-line specifies the time warp and individual machines may vary in the magnitude of time warp supported, correlated to the machines raw computing capacity.

29 uFldBeaconRangeSensor: Simulating Vehicle to Beacon Ranges

The [uFldBeaconRangeSensor](#) application is a tool for simulating an on-board sensor that provides a range measurement to a beacon where either (a) the vehicle knows its own position but is trying to determine the position of the beacon via a series of range measurements, or (b) the vehicle knows where the beacon(s) are but is trying to determine its own position based on the range measurements from one or more beacons at known locations.

The range-only sensor may be one that responds to a query, e.g., an acoustic ping, with an immediate reply, e.g. another acoustic ping or echo, from which the range from the source to the beacon is determined by the time-of-flight of the message through the medium, e.g., the approximate speed of sound through water. This idea is shown below on the left. Alternatively, if the beacon emits its message on a precise schedule with a clock precisely synchronized with the vehicle clock, the range measurement may be derived without requiring a separate query from the vehicle. This is the idea behind long baseline acoustic navigation, [4–7]. This idea is shown below on the right.

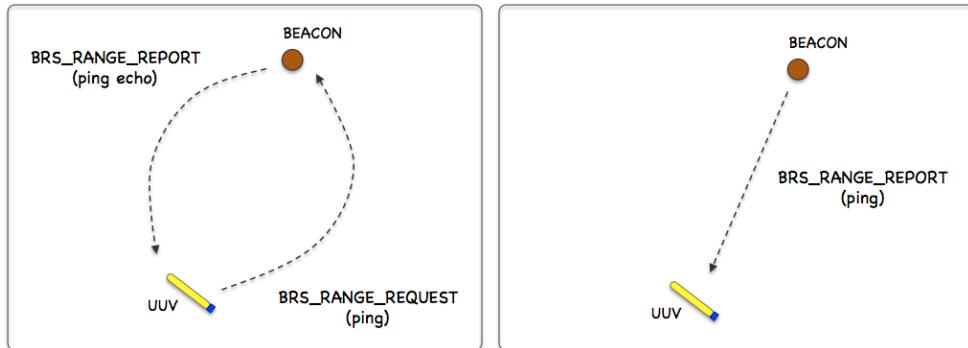


Figure 105: **Beacon Range Sensors:** A vehicle determines its range to a beacon by either (a) emitting a query and waiting for a reply, or (b) waiting for a message to be emitted on fixed schedule. In each case, the time-of-flight of the message through the medium is used to calculate the range.

In the [uFldBeaconRangeSensor](#) application, the beacon and vehicle locations are known to the simulator, and a tidy [BRS_RANGE_REPORT](#) message is sent to the vehicle(s) as a proxy to the actual range sensor and calculations that would otherwise reside on the vehicle. The MOOS app may be configured to have beacons provide a range report either (a) solicited with a range request, or (b) unsolicited. One may also configure the range at which a range request will be heard, and the range at which a range report will be heard. The app may be further configured to either (1) include the beacon location and ID, or (2) not include the beacon location or ID.

Typical Simulator Topology

The typical module topology is shown in Figure 106 below. Multiple vehicles may be deployed in the field, each periodically communicating with a shoreside MOOS community running a single instance of [uFldBeaconRangeSensor](#). Each vehicle regularly sends a node report noted by the simulator to keep an updated calculation of each vehicle to each simulated beacon. When a beacon wants to simulate a ping, or range request, it generates the [BRS_RANGE_REQUEST](#) message send to the shore. After the simulator calculates the range, a reply message, [BRS_RANGE_REPORT](#) is sent to the vehicle.

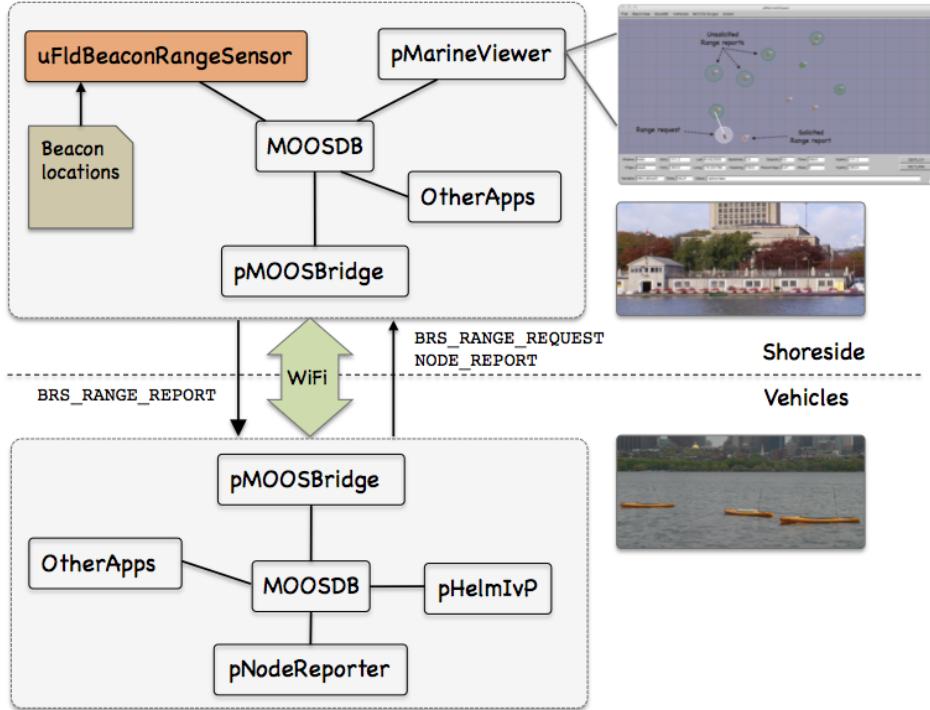


Figure 106: **Typical `uFldBeaconRangeSensor` Topology:** The simulator runs in a shoreside computer MOOS community and is configured with the beacon locations. Vehicles accessing the simulator periodically send node reports to the shoreside community. The simulator maintains a running estimate of the range between vehicles and beacons, modulo latency. A vehicle simulates a ping by sending a range request to shore and receiving a range report in return from the simulator.

If running a pure simulation (no deployed vehicles), both MOOS communities may simply be running on the same machine configured with distinct ports. The `pShare` application is shown here for communication between MOOS communities, but there are other alternatives for inter-community communication and the operation of `uFldBeaconRangeSensor` is not dependent on the manner of inter-communication communications.

29.1 The `uFldBeaconRangeSensor` Interface and Configuration Options

The `uFldBeaconRangeSensor` application may be configured with a configuration block within a `.moos` file. Its interface is defined by its publications and subscriptions for MOOS variables consumed and generated by other MOOS applications. An overview of the set of configuration options and interface is provided in this section.

29.1.1 Configuration Parameters of `uFldBeaconRangeSensor`

The following parameters are defined for `uFldBeaconRangeSensor`. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so.

Listing 29.1: Configuration Parameters for `uFldBeaconRangeSensor`.

beacon: Description of beacon location and properties. Section [29.3.1](#).
default_beacon_freq: Frequency of unsolicited beacon broadcasts ("never"). Section [29.3.2](#).
default_beacon_report_range: Range at which a vehicle will hear a range report (100). Section [29.3.1](#).
default_beacon_width: Width of beacons (meters) when rendered (4). Section [29.3.1](#).
default_beacon_color: Color of beacons when rendered ("red"). Section [29.3.1](#).
default_beacon_shape: Shape of beacons when rendered ("circle"). Section [29.3.1](#).
ping_payments: How pings treated w.r.t. ping wait time ("upon_response"). Section [29.3.4](#).
ground_truth: If true, ground truth is also reported when noise is added. Section [29.3.5](#).
ping_wait: Mandatory number of seconds between successive vehicle pings. Section [29.3.4](#).
reach_distance: Range at which a vehicle ping will be heard (100).
report_vars: Determines variable name(s) used for range report ("short").
rn_algorithm: Algorithm for adding random noise to range measurements.
verbose: If true, verbose status message terminal output (false).

An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command line:

```
$ uFldBeaconRangeSensor --example or -e
```

This will show the output shown in Listing 2 below.

Listing 29.2: Example configuration of the `uFldBeaconRangeSensor` application.

```

1 =====
2 uFldBeaconRangeSensor Example MOOS Configuration
3 =====
4 Blue lines: Default configuration
5 Magenta lines: Non-default configuration
6
7 {
8   AppTick    = 4
9   CommsTick = 4
10
11 // Configuring aspects of vehicles in the sim
12 reach_distance = default = 200 // or {nolimit}
13 reach_distance = henry = 40 // meters
14 ping_wait      = default = 30 // seconds
15 ping_wait      = henry   = 120
16 ping_payments  = upon_response // or {upon_receipt, upon_request}
17

```

```

18 // Configuring manner of reporting
19 report_vars = short // or {long, both}
20 ground_truth = true // or {false}
21 verbose = true // or {false}
22
23 // Configuring default beacon properties
24 default_beacon_shape = circle // or {square, diamond, etc.}
25 default_beacon_color = orange // or {red, green, etc.}
26 default_beacon_width = 4
27 default_beacon_report_range = 100
28 default_beacon_freq = never // or [0,inf]
29
30 // Configuring Beacon properties
31 beacon = x=200, y=435, label=01, report_range=45
32 beacon = x=690, y=205, label=02, freq=90
33 beacon = x=350, y=705, label=03, width=8, color=blue
34
35 // Configuring Artificial Noise
36 rn_algorithm = uniform,pct=0 // pct may be in [0,1]
37 }

```

29.2 Publications and Subscriptions for uFldBeaconRangeSensor

The interface for `uFldBeaconRangeSensor`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldBeaconRangeSensor --interface or -i
```

29.2.1 Variables Published by uFldBeaconRangeSensor

The primary output of `uFldBeaconRangeSensor` to the MOOSDB is posting of range reports, visual cues for the range reports, and visual cues for the beacons themselves.

- **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 29.3.6.
- **BRS_RANGE_REPORT**: A report on the range from a particular beacon to a particular vehicle.
- **BRS_RANGE_REPORT_NAMEJ**: A report on the range from a particular beacon to vehicle NAMEJ.
- **VIEW_MARKER**: A description for visualizing the beacon in the field. (Section 29.4)
- **VIEW_RANGE_PULSE**: A description for visualizing the beacon range report. (Section 29.4)

The range report format may vary depending on user configuration. Some examples:

```

BRS_RANGE_REPORT = "name=alpha,range=129.2,time=19473362764.169"
BRS_RANGE_REPORT = "name=alpha,range=129.2,id=23,x=54,y=90,time=19473362987.428"
BRS_RANGE_REPORT_ALPHA = "range=129.2,time=19473362999.761"

```

The vehicle name may be embedded in the MOOS variable name to facilitate distribution of report messages to the appropriate vehicle with `pShare`.

29.2.2 Variables Subscribed for by uFldBeaconRangeSensor

The `uFldBeaconRangeSensor` application will subscribe for the following four MOOS variables:

- `APPCAST_REQ`: A request to generate and post a new apppcast report, with reporting criteria, and expiration. Section 11.10.4.
- `BRS_RANGE_REQUEST`: A request to generate range reports for all beacons to all vehicles within range of the beacon.
- `NODE_REPORT`: A report on a vehicle location and status.
- `NODE_REPORT_LOCAL`: A report on a vehicle location and status.

Command Line Usage of uFldBeaconRangeSensor

The `uFldBeaconRangeSensor` 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:

```
$ uFldBeaconRangeSensor --help or -h
```

Listing 29.3: Command line usage for the `uFldBeaconRangeSensor` tool.

```
1 Options:
2   --alias=<ProcessName>
3     Launch uFldBeaconRangeSensor with the given process
4     name rather than uFldBeaconRangeSensor.
5   --example, -e
6     Display example MOOS configuration block.
7   --help, -h
8     Display this help message.
9   --interface, -i
10    Display MOOS publications and subscriptions.
11   --version,-v
12    Display release version of uFldBeaconRangeSensor.
13   --verbose=<setting>
14    Set verbosity. true or false (default)
15
16 Note: If argv[2] does not otherwise match a known option,
17       then it will be interpreted as a run alias. This is
18       to support pAntler launching conventions.
```

29.3 Using and Configuring uFldBeaconRangeSensor

The `uFldBeaconRangeSensor` application is configured primarily with a set of beacons, and a policy for generating range reports to one or more simulated vehicles. The reports may be sent to the vehicles upon a query (solicited) or may be sent unsolicited based on a configured broadcast schedule for each beacon. The possible simulator configuration arrangements are explored by first considering a simple case shown in Figure 107 below, representing a vehicle navigating with three beacons.

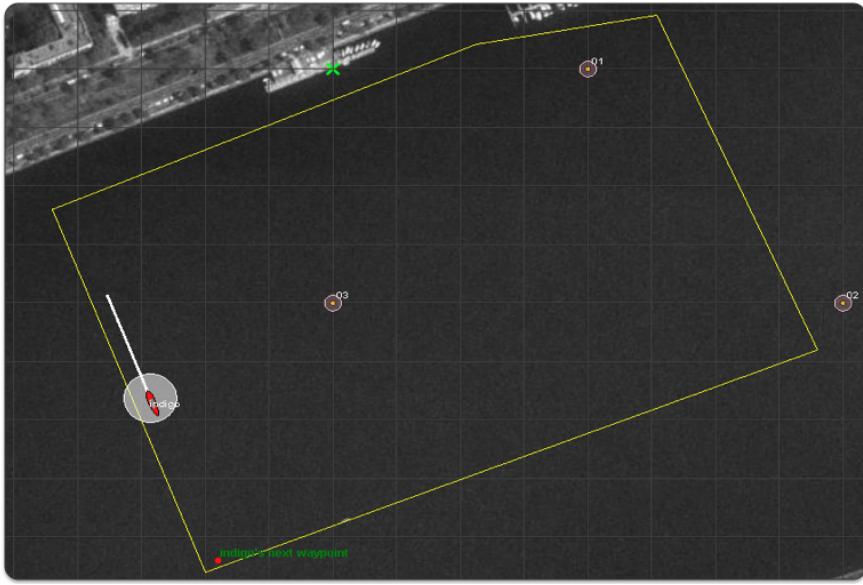


Figure 107: **Simulated LBL Beacons:** Three beacons are simulated, labelled 01, 02, and 03. The vehicle periodically issues a query to which the beacons immediately reply. The `uFldBeaconRangeSensor` application handles the queries and generates the range reports sent to each vehicle. The growing circles rendered around the vehicle and beacons represent the generation of the range query and range reports respectively.

The configuration for the `uFldBeaconRangeSensor` is shown in Listing 4 below. The three beacons are configured in lines 19-21. The configuration on line 7 indicates that a beacon query will be heard regardless of the range between the vehicle and the beacon. In the other direction, the range report from the beacon will only be heard if the vehicle is within 100 meters. Line 8 indicates that ping or range request will be honored by the simulator at most once every 30 seconds, and this clock is reset each time a range request is honored by the simulator with the configuration on line 9.

Listing 29.4: Example configuration of the `uFldBeaconRangeSensor` application.

```

1 ProcessConfig = uFldBeaconRangeSensor
2 {
3     AppTick      = 4    // Standard MOOSApp configurations
4     CommsTick   = 4
5
6     // Configuring aspects of the vehicles
7     reach_distance = default = nolimit
8     ping_wait      = 30
9     ping_payments  = upon_accept
10
11    report_vars   = short
12
13    default_beacon_freq  = never      // Only on request (ping)
14    default_beacon_shape = circle
15    default_beacon_color = orange
16    default_beacon_width = 5
17    default_beacon_report_range = 100

```

```

18
19   beacon = label=01, x=200, y=0
20   beacon = label=02, x=400, y=-200
21   beacon = label=03, x=0,   y=-200, color=red, shape=triangle, report_range=80
22 }

```

The three beacons in this example are configured on lines 19-21 with unique labels and locations. Each beacon has additional properties, such as its shape, color and width when rendered. Default values for these properties are given in lines 14-16, but may be overridden for a particular beacon as on line 22.

The key configuration line in this example is on line 13 which indicates the beacons by default never generate an unsolicited range report. Reports are only generated upon request. In this example, the simulated vehicle would receive successive groups of **BRS_RANGE_REPORT** postings from all three simulated beacons, each time the vehicle posts a **BRS_RANGE_REQUEST** message to the MOOSDB. This example is runnable in the *indigo* example mission distributed with the MOOS-IvP source code and described a bit later in Section 29.5.

29.3.1 Configuring the Beacon Locations and Properties

One or more beacons may be configured by the **beacon** configuration parameter provided in the **uFldBeaconRangeSensor** configuration block of the .moos file. Each beacon is configured with a line:

```
beacon = <configuration>
```

The **<configuration>** component is a comma-separated list of **parameter=value** pairs, with the following possible parameters: **x**, **y**, **label**, **freq**, **shape**, **width**, **color**, and **query_range**. The following are typical examples:

```

beacon = x=200, y=260, label=03, freq=10
beacon = x=-40, y=150, label=04, freq=5:15, color=red, shape=circle, width=4, report_range=200

```

The **x** and **y** parameters specify the beacon locations in local coordinates. Like several other MOOS applications, the **uFldBeaconRangeSensor** app looks for a global parameter in the .moos configuration file naming the position of the datum, or 0,0 position in latitude, longitude coordinates. The **label** parameter provides a unique identifier for the beacon. If a beacon entry is provided using a previously used label, the new beacon will overwrite the prior beacon in the simulator. If no label is provided, an automatic label will be generated equivalent to the index of the new beacon. The **freq** parameter specifies, in seconds, how often *unsolicited* range reports are generated for each beacon. A simple numerical value may be given, or a colon-separated pair of values as shown above may be used to specify a uniformly random interval of possible durations. The duration between posts will be reset after each post. The **report_range** parameter specifies the distance, in meters, that a vehicle must be to hear a range report generated by a beacon. The **shape** parameter indicates the shape used by applications like **pMarineViewer** when rendering the beacon. The **uFldBeaconRangeSensor** application generates a **VIEW_MARKER** post to the MOOSDB for each beacon, once upon startup of the simulator. The **VIEW_MARKER** structure and possible shapes are described in the **pMarineViewer** section in [2]. The **color** parameter specifies the color to be used when rendering the beacon. Legal

color strings are described in Appendix B. The `width` parameter is used to indicate the width, in meters, when rendering the beacon.

For convenience, default values for several of the above properties may be provided with the following five supported configuration parameters:

```
default_beacon_report_range = 100
default_beacon_shape = circle
default_beacon_color = orange
default_beacon_width = 4
default_beacon_freq = never
```

The above configuration values also represent all the default values. A beacon configuration that includes any of the above five parameters explicitly, will override any default values.

29.3.2 Unsolicited Beacon Range Reports

The simulator may be configured to have all its beacons periodically generate a range report to all vehicles within range. The schedule of reporting may be uniform across all beacons, or individually set for each beacon. The interval of time between reports may also be set to vary according to a uniformly random time interval. By default, beacons are configured to never generate unsolicited range reports unless their frequency parameter is set to something else besides the default value of "never". The default value for all beacons may be configured with the following parameter in the `uF1dBeaconRangeSensor` configuration block with the following:

```
default_beacon_freq = 120
```

The parameter value is given in seconds. To configure an interval to vary randomly on each post within a given range, e.g., somewhere between one and two minutes, the following may be used instead:

```
default_beacon_freq = 60:120
```

To configure the simulator to never generate an unsolicited range report, i.e., only solicited reports, use the following:

```
default_beacon_freq = never
```

Upon each range report post to the MOOSDB, the interval until the next post is recalculated. The beacon schedule may also be configured to be unique to a given beacon. The beacon configuration line accepts the `freq` parameter as described earlier in Section 29.3.1. The configuration provided for an individual beacon overrides the default frequency configuration.

Once a beacon has generated a report, it will not generate another *unsolicited* report until after the prevailing time interval has passed. However, if the simulator detects that the beacon has been solicited for a range report via an explicit range request from a nearby vehicle, a range report may be generated immediately. In this case the clock counting down to the beacon's next unsolicited report is reset.

29.3.3 Solicited Beacon Range Reports

The `uFldBeaconRangeSensor` application accepts requests from vehicles, and may or may not generate one or more range reports for beacons within range of the vehicle making the request. In short, things operate like this: (a) a range request is received by `uFldBeaconRangeSensor` through its mailbox on the variable `RANGE_REQUEST`, (b) a determination is made as to whether the request is within range of the beacon and whether the request is allowed based on limits on the frequency of range requests, (c) a range report is generated and posted to the variable `BRS_RANGE_REPORT`. The following is an example of the range request format:

```
BRS_RANGE_REQUEST = "name=charlie"
```

Note that if a vehicle generates a range request triggering a range report from a beacon, the range report is sent to *all* vehicles within range of the beacon. Presumably the simulator has also received, at some point in the past, a node report, typically generated from the `pNodeReporter` application (described in the `pNodeReporter` section in [2]) running on the vehicle. So the simulator not only knows which vehicle is making the range request, but also where that vehicle is located. It needs the vehicle location to determine the range between the vehicle and beacon, to generate the requested range report. The simulator also uses this range information to decide if it wants regard the beacon as being close enough to hear the request, and whether the vehicle is close enough to the beacon to hear the report.

29.3.4 Limiting the Frequency of Vehicle Range Requests

From the perspective of operating a vehicle, one may ask: why not request a range report from all beacons as often as possible? There may be reasons why this is not feasible outside simulation. Limits may exist due to power budgets of the vehicle and/or beacons, and there may be prevailing communications protocols that make it at least impolite to be pushing range requests through a shared communications medium.

To reflect this limitation, the `uFldBeaconRangeSensor` application may be configured to limit the frequency in which a vehicle's range request (or ping) will be honored with a range report reply. By default this frequency is set to once every 30 seconds for all vehicles. The default for all vehicles may be changed with the following configuration in the `.moos` file:

```
ping_wait = default = 60
```

If the time interval as above is set to 60 seconds, what happens if a vehicle requests a range report 40 seconds after its previous request? Is it simply ignored, needing to wait another 20 seconds? Or is the clock reset to zero forcing the vehicle to wait 60 seconds before a ping is honored? By default, the former is the case, but the simulator may be configured to the more draconian option with:

```
ping_payments = upon_request
```

Suppose the minimum time interval has elapsed, but the querying/pinging vehicle is too far out of range from any beacon to hear even a single range report. Will the result be that the clock is reset

to zero, forcing the vehicle to wait another 60 seconds before a query is honored? By default this is the case, but the simulator may be configured to not reset the clock unless the querying vehicle has received at least one range report for its query:

```
ping_payments = upon_response
```

In short, the simulator configuration parameter, `ping_payments`, may be configured with one of three options, "upon_request", "upon_response", or "upon_accept", with the default being the latter.

29.3.5 Producing Range Measurements with Noise

In the default configuration of `uFldBeaconRangeSensor`, range reports are generated with the most precise range estimate as possible, with the only error being due to the latency of the communications generating the range request and range report. Additional noise/error may be added in the simulator for each range report with the following configuration parameter:

```
rn_algorithm = uniform,pct=0.12 // Values in the range [0,1]
```

Currently the only noise algorithm supported is the generation of uniformly random noise on the range measurement. The noise level, θ , set with the parameter `rn_uniform_pct`, will generate a noisy range from an otherwise exact range measurement r , by choosing a value in the range $[\theta r, r + \theta r]$. The range without noise, i.e., the ground truth, may also be reported by the simulator if desired by setting the configuration parameter:

```
ground_truth = true
```

This will result in an additional MOOS variables posted, `BRS_RANGE_REPORT_GT`, with the same format as `BRS_RANGE_REPORT`, except the reported range will be given without noise.

29.3.6 Terminal and AppCast Output

The `uFldBeaconRangeSensor` application produces some useful information to the terminal and identical content through appcasting. An example is shown in Listing 5 below. On line 2, the name of the local community or vehicle name is listed on the left. On the right, "0/0(1478) indicates there are no configuration or run warnings, and the current iteration of `uFldBeaconRangeSensor` is 1478.

Lines 4-12: Beacon Configuration

Lines 4-12 convey the prevailing beacon configuration settings. They may reflect the default values or values read from the configuration block like the one shown previously in Listing 2. Each beacon has an ID (first column), and the location in local coordinates in the next two columns. The fourth column indicates whether reports are generated in response to a poll, i.e., range request, or on a regular timed cycle as in beacon #03. If the latter type, the fifth column indicates the number of unsolicited pings generated. The *Pings Recvd* and *Pings Gen'd* columns show the number of range requests received and generated. *Noise* column indicates if there is any noise applied to the reports. The last two columns show the push and pull distances.

Listing 29.5: Example `uFldBeaconRangeSensor` console or appcast output.

```

1 =====
2 uFldBeaconRangeSensor hotel                               0/0(1478)
3 =====
4 Beacons(5):
5          Pings  Pings  Pings      Push  Pull
6 ID     X      Y  Freq Unsol Recvd Gen'd Noise Dist  Dist
7 ---  ----  ----  ----  ----  ----  ----  ----  ----  ----
8 05   115  -150  poll  0     0     0     0    33    95
9 04   -65  -345  poll  0    16    16    0    85    75
10 03    0  -200  15.0  46    16    16    0    85    75
11 02  -100  -100  poll  0    14    14    0    85    75
12 01   200    0  poll  0     0     0     0    85    75
13
14 Vehicles(1):
15      Push  Pull  Unsol  Pings  Pings  Too  Too
16 VName  Dist  Dist  Rec'd  Gen'd  Rep'd  Freq  Far
17 -----  ----  ----  -----  -----  -----  ----  -----
18 hotel  100  100  22    24    46    0    74
19
20 =====
21 Most Recent Events (8):
22 =====
23 [864.51]: Beacon[03]  ))--> Range Broadcast to HOTEL
24 [849.27]: Beacon[03]  --> Range Reply to HOTEL
25 [849.27]: Beacon[03] <---- Ping from HOTEL
26 [849.27]: Beacon[04]  --> Range Reply to HOTEL
27 [849.27]: Beacon[04] <---- Ping from HOTEL
28 [849.27]: Range Request resets the clock for vehicle HOTEL
29 [849.27]: Valid Range Request from HOTEL. Checking beacon ranges...
30 [836.27]: Beacon[03]  ))--> Range Broadcast to HOTEL

```

Unless the verbose setting is turned on, the output ending on line 37 above should be the last output written to the console for the duration of the simulator.

In the verbose mode, the simulator will produce event-based output as shown in the example above beginning on line 38. The asterisks in lines 39, 46, and 50 are not merely visual separators. An asterisk represents a single receipt of a `NODE_REPORT` message. Receiving node reports is essential for the operation of the simulator and this provides a bit of visual verification that this is occurring. Presumably node reports are being received much more often than range requests and range reports are handled, as is the case in the above example. The first time a node report is received for a particular vehicle, an announcement is made as shown on line 38.

In addition to handling incoming node reports, on any given iteration, the simulator may also handle an incoming range request, or may generate an unsolicited range report based on a beacon schedule. Console output for incoming range requests may look like that shown in lines 40-45 above. First the range request and the requesting vehicle is announced as on line 40. The elapsed time since the vehicle last made a range request is shown as on line 41. If the request is honored by the simulator, this is indicated as shown on line 42. Otherwise a reason for denial may be shown. If the query is accepted, range reports may be generated for one or more vehicles. For each such vehicle, a

line announcing the new report is generated, as in lines 43-45. On an iteration where unsolicited range reports are generated, output similar to that shown in lines 47-49 will be generated. For each report, the beacon and receiving vehicle are named.

29.4 Interaction between uFldBeaconRangeSensor and pMarineViewer

The `uFldBeaconRangeSensor` application will post certain messages to the MOOSDB that may be subscribed for by GUI based applications like `pMarineViewer` for visualizing the posting of `BRS_RANGE_REPORT` and `BRS_RANGE_REQUEST` messages, as well as visualizing the beacon locations. A snapshot of the `pMarineViewer` window is shown below, with one vehicle and several beacons.

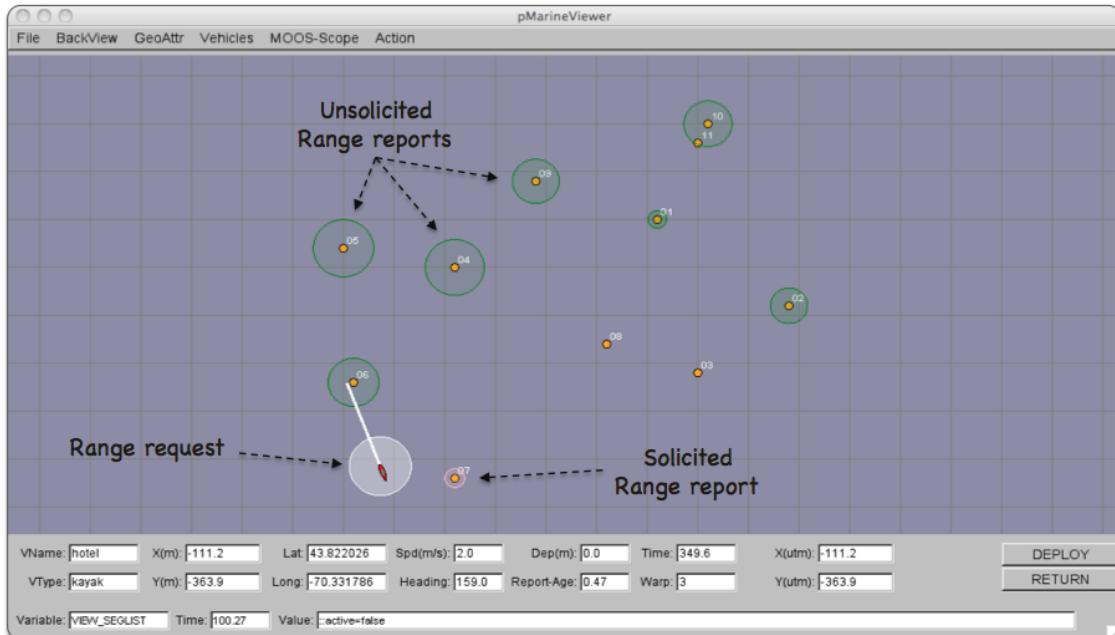


Figure 108: **Beacons in the pMarineViewer:** The `VIEW_RANGE_PULSE` message is passed to pMarineViewer to render unsolicited range reports (here in green), range requests from a vehicle (here in white), and solicited range reports in response to a range request (here in pink). The viewer also renders the beacons and their labels upon receiving `VIEW_MARKER` messages posted by the `uFldBeaconRangeSensor` application. The pulses are only momentarily visible until another `VIEW_RANGE_PULSE` message is received.

29.4.1 The `VIEW_MARKER` Data Structure

The `uFldBeaconRangeSensor` application, upon startup, posts the beacon locations in the form of the `VIEW_MARKER` data structure. This MOOS variable is one of the default variables registered for by the `pMarineViewer` application. The types of supported markers are described in the `pMarineViewer` section in [2]. The marker type, size and color are configurable in the `uFldBeaconRangeSensor` configuration block. The user may use the variation in marker rendering to correspond to variation in beacon reporting or querying characteristics.

29.4.2 The VIEW_RANGE_PULSE Data Structure

A *range pulse* message is used by the `uFldBeaconRangeSensor` application to convey visually the generation of a range report, or the receipt of a range request. The pulse is rendered as a ring with a growing radius having either the beacon or the vehicle at the center. A pulse emanating from a beacon indicates a range report, and a pulse emanating from a vehicle indicates a range request. By default different colors may be used for solicited and unsolicited range reports. In Figure 108 for example, the green rings represent unsolicited reports, the white ring represents a range request made by the vehicle, and the pink ring represents a response to the range request made by the one beacon within range to the vehicle.

The `VIEW_RANGE_PULSE` message is a data structure implemented in the `XYRangePulse` class, with the following format by example:

```
VIEW_RANGE_PULSE = x=-40,y=-150,radius=40,duration=15,fill=0.25,fill_color=green,  
label=04,edge_color=green,time=3892830128.5,edge_size=1
```

The `x` and `y` parameters convey the center of the pulse. The `radius` parameter indicates the radius of the circle at its maximum. The `duration` parameter is the number of seconds the pulse will be rendered. The pulse will grow its radius linearly from zero meters at zero seconds to `radius` meters at `duration` seconds. The `fill` parameter is in the range $[0, 1]$, where 0 is full transparency (clear) and 1 is fully opaque. The pulse transparency increases linearly as the range ring is rendered. The starting transparency at $radius = 0$ is given by the `fill` parameter. The transparency at the maximum radius is zero. The `fill_color` parameter specifies the color rendered to the internal part of the range pulse. The choice of legal colors is described in Appendix B. The `label` is a string that uniquely identifies the range instance to consumers like `pMarineViewer`. As with other objects in `pMarineViewer`, if it receives an object the same label and type as one previously received, it will replace the old object with the new one in its memory. The `edge_color` parameter describes the color of the ring rendered in the range pulse. The `edge_size` likewise describes the line width of the rendered ring. The `time` parameter indicates the UTC time at which the range pulse object was generated.

29.5 The Indigo Example Mission Using `uFldBeaconRangeSensor`

The *indigo* mission is distributed with the MOOS-IvP source code and contains a ready example of the `uFldBeaconRangeSensor` application, configured with three beacons acting as long baseline (LBL) beacons as described at the beginning of Section 29.3. Assuming the reader has downloaded the source code available at www.moos-ivp.org and built the code according to the discussion in Section 1.2.9, the *indigo* mission may be launched by:

```
$ cd moos-ivp/ivp/missions/s9_indigo/  
$ ./launch.sh 10
```

The argument, 10, in the line above will launch the simulation in 10x real time. Once this launches, the `pMarineViewer` GUI application should launch and the mission may be initiated by hitting the DEPLOY button. The vehicle will traverse a survey pattern over the rectangular operation region

shown in Figure 107, periodically generating a range request to the three beacons. With each range request, a white range pulse should be visible around the vehicle. Almost immediately afterwards, a range report for each beacon is generated and a red range pulse around each beacon is rendered. The snapshot in Figure 107 depicts a moment in time where the range report visual pulses are beginning to grow around the beacons and the range request visual pulse is still visible around the one vehicle.

How does the simulated vehicle generate a range request? In practice a user may implement an intelligent module to reason about when to generate requests, but in this case the `uTimerScript` application is used by creating a script that repeats endlessly, generating a range request once every 25-35 seconds. The script is also conditioned on `(NAV_SPEED > 0)`, so the pinging doesn't start until the vehicle is deployed. The configuration for the script can be seen in the `uTimerScript` configuration block in the `indigo.moos` file. More on the `uTimerScript` application can be found in Section 6.

29.5.1 Examining the Log Data from the Indigo Mission

After the launch script above has launched the simulation, the script should leave the console user with the option to "Exit and Kill Simulation" by hitting the '2' key. Once the vehicle has been deployed and traversed to one's satisfaction, exit the script. A log directory should have been created by the `pLogger` application in the directory where the simulation was launched. The directory name should begin with `MOOSLog_` with the remainder of the directory name composed from the timestamp of the launch.

Let's take a look at some of the data related to the simulation and `uFldBeaconRangeSensor` in particular. A dump of the entire file reveals a deluge of information. To look at the information relevant to the `uFldBeaconRangeSensor` application, the file is pruned with the `aloggrep` tool:

```
$ aloggrep MOOSLog_1_2_2011_____2_32_48.alog uFldBeaconRangeSensor uTimerScript
```

This produces a subset of the alog file similar to that shown in Listing 6, showing only log entries made by either the `uFldBeaconRangeSensor` application, or the `uTimerScript` application which generated all the range requests as described above. The first three posts made to the MOOSDB by `uFldBeaconRangeSensor` are the `VIEW_MARKER` posts representing a visual cue for the `pMarineViewer` application to render the three beacons.

Listing 29.6: A subset of the data logged from the Indigo example mission's alog file.

```
%%%%%
%% LOG FILE:      ./MOOSLog_22_2_2011_____17_27_06/MOOSLog_22_2_2011_____17_27_06.alog
%% FILE OPENED ON Tue Feb 22 17:27:06 2011
%% LOGSTART          23371445284.8
%%%%%
55.697  VIEW_MARKER    uFldBeaconRangeSensor  x=0,y=-200,label=03,color=orange,type=circle,width=4
55.698  VIEW_MARKER    uFldBeaconRangeSensor  x=400,y=-200,label=02,color=orange,type=circle,width=4
55.698  VIEW_MARKER    uFldBeaconRangeSensor  x=200,y=0,label=01,color=orange,type=circle,width=4
100.663 BRS_RANGE_REQUEST  uTimerScript      name=indigo
100.846  VIEW_RANGE_PULSE  uFldBeaconRangeSensor  x=-97.65,y=-64.84,radius=50,duration=6,...
```

```

100.846 BRS_RANGE_REPORT uFldBeaconRangeSensor vname=indigo,range=166.7446,id=03,time=23371445385.6
100.846 VIEW_RANGE_PULSE uFldBeaconRangeSensor x=0,y=-200,radius=40,duration=15,...
100.846 BRS_RANGE_REPORT uFldBeaconRangeSensor vname=indigo,range=515.6779,id=02,time=23371445385.6
100.846 VIEW_RANGE_PULSE uFldBeaconRangeSensor x=400,y=-200,radius=40,duration=15,...
100.847 BRS_RANGE_REPORT uFldBeaconRangeSensor vname=indigo,range=304.6305,id=01,time=23371445385.6
100.847 VIEW_RANGE_PULSE uFldBeaconRangeSensor x=200,y=0,radius=40,duration=15,...
160.419 BRS_RANGE_REQUEST uTimerScript name=indigo
160.597 VIEW_RANGE_PULSE uFldBeaconRangeSensor x=-197.96,y=-129.16,radius=50,duration=6,...
160.597 BRS_RANGE_REPORT uFldBeaconRangeSensor vname=indigo,range=210.2533,id=03,time=23371445445.3
160.597 VIEW_RANGE_PULSE uFldBeaconRangeSensor x=0,y=-200,radius=40,duration=15,...
160.597 BRS_RANGE_REPORT uFldBeaconRangeSensor vname=indigo,range=602.1416,id=02,time=23371445445.3
160.597 VIEW_RANGE_PULSE uFldBeaconRangeSensor x=400,y=-200,radius=40,duration=15,...
160.597 BRS_RANGE_REPORT uFldBeaconRangeSensor vname=indigo,range=418.3951,id=01,time=23371445445.3
160.597 VIEW_RANGE_PULSE uFldBeaconRangeSensor x=200,y=0,radius=40,duration=15,...

```

The first range request is generated at time 100.663 by the `uTimerScript`. The `uFldBeaconRangeSensor` application receives this mail and posts a range pulse at time 100.846 conveying the range request from the vehicle, e.g., the white circle in Figure 107. The range request is met immediately and two posts are generated for each beacon. The `VIEW_RANGE_PULSE` message indicates the simulator has generated a range report for the beacon (the red circle in Figure 107). The `BRS_RANGE_REPORT` message is the actual range report to be used by the vehicle as it sees fit.

29.5.2 Generating Range Report Data for Matlab

The log files generated as in Listing 6 above may be processed to form a table of values suitable for Matlab processing. The `alog2rng` tool may be run on an alog file from the command line:

```
$ alog2rng MOOSLog_21_2_2011_____22_32_48.alog
```

This will generate a table of data like that below. The left column is the timestamp from the log file. The next N columns are the range measurements from each beacon. And the last three columns are the "ground truth" vehicle position and heading. The last three columns may be excluded with the `--nav=false` switch on the command line.

Time	03	02	01	NAV_X	NAV_Y	NAV_HDG
100.846	166.7446	515.6779	304.6305	-99.371	-65.917	238.000
160.597	210.2533	602.1416	418.3951	-198.538	-130.397	197.456
224.844	163.4205	558.5143	433.6937	-155.744	-248.991	159.000

The `alog2rng` tool is part of the Alog-Toolbox along with other tools for examining and modifying alog files generated by `pLogger`.

30 uFldContactRangeSensor: Detecting Contact Ranges

30.1 Overview

The `uFldContactRangeSensor` application is a tool for simulating range measurements to off-board contacts, as a proxy for an on-board active sonar sensor. The range-only measurements are provided conditionally, depending on the range between the pinging vehicle and the contact. The simulator may optionally be configured to provide range measurements with noise.

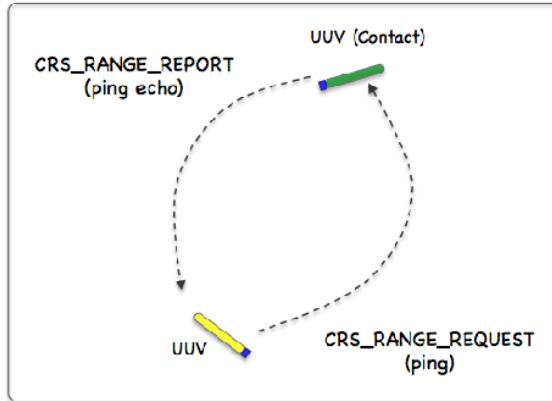


Figure 109: **Simulated Active Sonar:** A vehicle determines its range to another vehicle by producing a simulated sonar ping (a range request to the simulator), and the simulator conditionally responds to the querying vehicle with a report containing the range to nearby vehicles. All vehicles send frequent and regular node reports to the simulator so the simulator can report the range between any two vehicles at any time. The simulator may or may not reply to the range request depending on the range between the two vehicles and thresholds configured by the user.

In the `uFldContactRangeSensor` application, the beacon and vehicle locations are known to the simulator, and a tidy `RANGE_REPORT` message is sent to the vehicle(s) as a proxy to the actual range sensor and calculations that would otherwise reside on the vehicle. The MOOS app may be configured to have beacons provide a range report either (a) solicited with a range request, or (b) unsolicited. One may also configure the range at which a range request will be heard, and the range at which a range report will be heard. The app may be further configured to either (1) include the beacon location and ID, or (2) not include the beacon location or ID.

30.2 Using uFldContactRangeSensor

30.2.1 Typical Topology

The typical module topology is shown in Figure 110 below. Multiple vehicles may be deployed in the field, each periodically communicating with a shoreside MOOS community running a single instance of `uFldContactRangeSensor`. Each vehicle regularly sends a node report noted by the simulator to keep an updated calculation of each vehicle to each simulated beacon. When a vehicle wants to simulate a ping, or range request, it generates the `CRS_RANGE_REQUEST` message sent to the shore. After the simulator calculates the range, a reply message, `CRS_RANGE_REPORT` message is sent to the vehicle, using `pShare` or similar app.

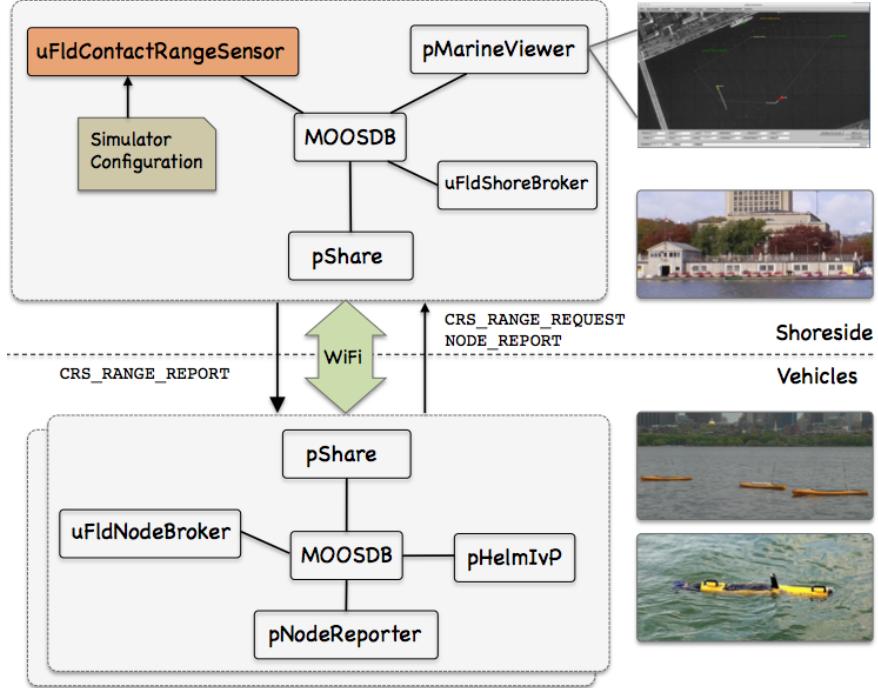


Figure 110: **Typical `uFldContactRangeSensor` Topology:** The simulator runs in a shoreside computer MOOS community. All deployed vehicles periodically send node reports to the shoreside community. The simulator maintains a running estimate of the range between vehicles, modulo latency. A vehicle simulates a ping by sending a range request to shore and receiving a range report in return from the simulator. The simulator also posts visual artifacts (`VIEW_RANGE_PULSE` messages) read by the `pMarineViewer` app optionally running shoreside.

If running a pure simulation (no physically deployed vehicles), both MOOS communities may simply be running on the same machine configured with distinct ports. The `pShare` application is shown here for communication between MOOS communities, but there are other alternatives for inter-community communication and the operation of `uFldContactRangeSensor` is not dependent on the manner of inter-communication communications.

30.2.2 Required MOOS Variable Bridges

Using `uFldContactRangeSensor` requires certain information flowing between the shoreside and vehicles communities as shown in Figure 110. From the vehicle to the shoreside, three variables need to be bridged. The below three lines should appear in the `uFldNodeBroker` configuration block on all vehicles.

```
// Bridges from Vehicle to Shoreside - in uFldNodeBroker configuration

bridge = src=APPCAST
bridge = src=NODE_REPORT_LOCAL, alias=NODE_REPORT
bridge = src=CRS_RANGE_REQUEST
```

The first two lines above would likely already be present due to their use in other applications. The `CRS_RANGE_REQUEST` is generated locally on the vehicle and sent to the shoreside community running

`uFldContactRangeSensor`. The above three lines may also be found in the Hugo example mission discussed in Section 30.7.

The below two lines should appear in the `uFldShoreBroker` configuration block in the shoreside MOOS community.

```
// Bridges from Shoreside to Vehicle - in uFldShore Broker configuration

bridge = src=APPCAST_REQ
bridge = src=CRS_RANGE_REPORT_$V, alias=CRS_RANGE_REPORT
```

The first line deals with appcasting and would likely be present anyway due to its use in other applications as well. The second line ensures that a separate bridge is made for all distinct known vehicles, as they become known to the shoreside. A range report posted by `uFldContactRangeSensor` to be sent to archie would be posted locally in the shoreside to the variable `CRS_RANGE_REPORT_ARCHIE` which would be sent only to the archie vehicle by `pShare`.

30.2.3 Range Requests and Range Reports

Range requests are made from vehicles to the simulator of the form:

```
CRS_RANGE_REQUEST = "name=archie"
```

The simulator only needs to know the requesting vehicle's name. It is also aware of the requesting and other vehicles' locations via incoming `NODE_REPORT` messages. When a range request is made, the simulator applies range and other criteria, discussed in the following sections. For a request from a vehicle named archie for example, the simulator may reply with a range report similar to:

```
CRS_RANGE_REPORT_ARCHIE = "range=126.54,target=jackal,time=19656022406.44"
```

If the `ground_truth` is set to true (the default), ground truth reports are also published of the form:

```
CRS_RANGE_REPORT_GT_ARCHIE = "range=127.12,target=jackal,time=19656022406.44"
```

These reports have the simulated range noise removed and are not typically sent to the vehicles. If the simulator is configured with `report_vars` set to "short", the above two posts are made with the following form instead:

```
CRS_RANGE_REPORT_ = "vname=archie,range=126.54,target=jackal,time=19656022406.44"
CRS_RANGE_REPORT_GT = "vname=archie,range=127.12,target=jackal,time=19656022406.44"
```

The name of the requesting vehicle is embedded in the report content rather than the MOOS variable. The default setting for `report_vars` is "long", producing the first style of reports above.

30.2.4 Configuring the Range Criteria

When a ping is received by the simulator, via `CRS_RANGE_REQUEST`, the simulator may or may not reply with an range report depending on the range between the two vehicles. The `uFldContactRangeSensor` simulator has two range configuration parameters:

```
reach_distance = default = <meters>    // or "nolimit"
reply_distance = default = <meters>    // or "nolimit"
```

Two separate range parameters are used so that the simulator can support scenarios where some vehicles have a more powerful sonar than others, and some targets are harder to detect (absorb or deflect more energy) than others. Both parameters are given in meters. If the user provides no configuration parameters, all vehicles will default to have the same reach and reply distances of 100 meters. The default values may be overridden with something like:

```
reach_distance = default = 120
reply_distance = default = 80
```

The above two lines, in effect, are the same as `reach_distance = 100` and `reply_distance = 100`. Things become interesting when individual vehicles are given values different from the default. Consider the more advantageously configured vehicle, *victor*, below:

```
reach_distance = victor = 250
reply_distance = victor = 20
```

If either the reach or reply distance for a given pair of vehicles is set to `nolimit`, then a range report will always be generated regardless of current range between the two vehicles. Future enhancements to this simulator module may include the factoring of vehicle speed and relative bearing to one another in the threshold determination of sending range reports.

30.2.5 Adding Exponentially Decaying Detection Probability Beyond Range

If one desires an additional probability to as to whether or not the pinging vehicle receives a ping from the pinged vehicle beyond the `reach_distance + reply_distance` specified in the MOOS configuration file. This is normally off unless configured in the application's configuration file. In order to use this feature, the following configuration parameter must be set:

```
exponential_decay = true =    // default is "false"
```

When the `exponential_decay` configuration parameter is turned on, and the vehicle requesting the range report is within the `reach_distance + reply_distance` range of the requested vehicle, this configuration will have no effect. However, if the vehicles are beyond this range, this assign a probability, which decays exponentially with increasing range a probability that a range report will still be received. The actual probability is calculated by the following equation:

$$Probability = e^{\left(\frac{3(Max - Range)}{Max} \right)} \quad (12)$$

where *Probability* is the probability that a range report will be received, *Max* is the `reach_distance + reply_distance`, and *Range* is the Range between the two vehicles. Figure 111 is a graphical representation of the probability of detection as a function of range when this configuration is turned on.

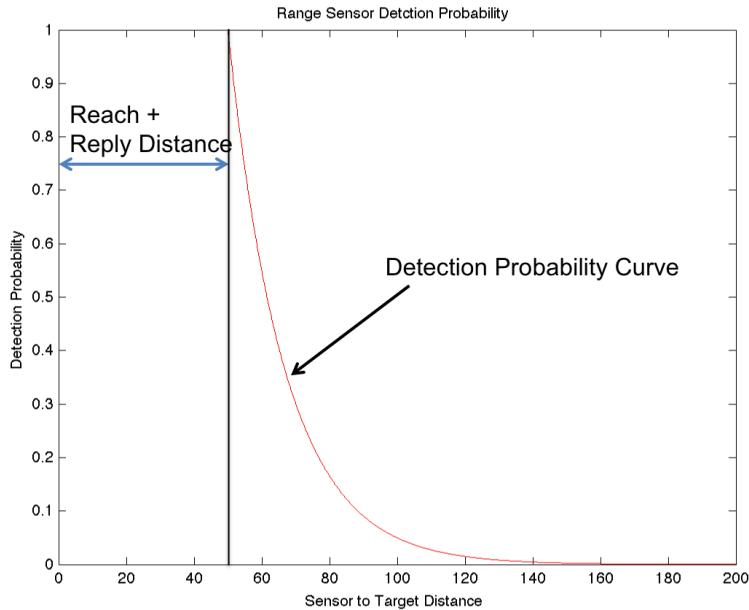


Figure 111: **Exponential Decay Configuration:** If the `exponential_decay` configuration parameter is set to true, the requesting vehicle will have an additional probability of detecting the requested vehicle even when beyond the `reach_distance + reply_distance` range, which is set to 50 in this example.

30.2.6 Limiting the Arcs of Vehicle Range Requests

This feature allows a vehicle to limit the range sensing capability to use specified arcs with respect to the relative heading of the vehicle. For instance a user could limit the sensor to only detect vehicles that are within 45 degrees of directly ahead, or 315 degrees relative to 45 degrees relative. Multiple sensor arcs can be specified, in one degree increments. The default is 360 degree coverage. To specify a range arc, two numbers must be specified, separated by a colon. The first number specifies the left side of the arc, and the second represents the right side of the arc. Valid values for relative headings are between 0 and 359 degrees. Multiple arcs can be separated by commas, or by multiple lines. Figure 112 shows some example arcs of where requesting vehicles can detect requested vehicles. Some example configurations are as follows:

```
sensor_arc = 360 // default, 360 degree coverage
sensor_arc = 315:45 // 90 degree coverage for front.
sensor_arc = 45:135,225:315 // two 90 degree arcs centered broadside
```

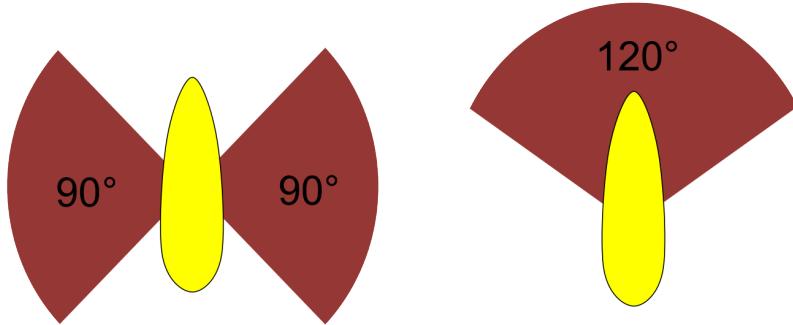


Figure 112: **Example Sensor Arc Configurations:** Two possible range sensor arc configurations.

30.2.7 Limiting the Frequency of Vehicle Range Requests

From the perspective of operating a vehicle, one may ask: why not request a range report (ping) as often as possible? There may be reasons why this is not feasible outside simulation. Limits may exist due to power budgets of the vehicle, and there may be prevailing protocols that make it at least impolite to be frequently pinging.

To reflect this limitation, the `uFldContactRangeSensor` utility may be configured to limit the frequency in which a vehicle's range request (or ping) will be honored with a range report reply. By default this frequency is set to once every 30 seconds for all vehicles. The default for all vehicles may be changed with the following configuration in the `.moos` file:

```
ping_wait = default = 60
```

The limits for a particular vehicle may be set with a similar configuration line:

```
ping_wait = henry = 90
```

30.2.8 Producing Range Measurements with Noise

In the default configuration of `uFldContactRangeSensor`, range reports are generated with the most precise range estimate as possible, with the only error being due to the latency of the communications generating the range request and range report. Additional noise/error may be added in the simulator for each range report with the following configuration parameter:

```
rn_algorithm = uniform,pct=0.12      // Values in the range [0,1]
```

or

```
rn_algorithm = gaussian,sigma=20      // Values in the range [0,inf]
```

For uniform random noise, the noise level, θ , set with the parameter `pct` subfield, will generate a noisy range from an otherwise exact range measurement r , by choosing a value in the range $[\theta r, r + \theta r]$. The range without noise, i.e., the ground truth, may also be reported by the simulator if desired by setting the configuration parameter:

```
ground_truth = true
```

This will result in an additional MOOS variables posted, `CRS_RANGE_REPORT_GT`, with the same format as `CRS_RANGE_REPORT`, except the reported range will be given without noise.

30.3 Configuration Parameters of `uFldContactRangeSensor`

The following parameters are defined for `uFldContactRangeSensor`. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so.

Listing 30.1: Configuration Parameters for `uFldContactRangeSensor`.

`sensor_arcs`: Set arcs relative to the heading of the vehicle in which the range sensor is able to detect vehicles.

An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command line:

```
$ uFldContactRangeSensor --example or -e
```

This will show the output shown in Listing 2 below.

Listing 30.2: Example configuration of the `uFldContactRangeSensor` application.

```
1 =====
2 uFldContactRangeSensor Example MOOS Configuration
3 =====
4
5 ProcessConfig = uFldContactRangeSensor
6 {
7     AppTick      = 4
8     CommsTick   = 4
9
10    // Configuring aspects of the vehicles in the sim
11    reach_distance = default = 100 // in meters or {nolimit}
12    reply_distance = default = 100 // in meters or {nolimit}
13    ping_wait      = default = 30 // in seconds
14
15    push_distance  = jackal  = 50
16    push_distance  = archie  = 190
17    ping_wait      = archie  = 32
18
19    // Configuring manner of reporting
20    report_vars    = short // or {long, both}
21    ground_truth   = true  // or {false}
22
23    // Configuring visual artifacts
24    ping_color     = white
25    reply_color    = chartreuse
26
27    // Configuring Artificial Noise
28    rn_algorithm   = uniform,pct=0.04
29
30    // Configuring Probability Detection with Exponential Decay
31    exponential_decay = false // default is false
32
33    // Configuring Sensor Arcs
34    sensor_arc    = 45:135,225:315 // for two side arcs
35    // sensor_arc = 315:45        // just for front
36    // sensor_arc = 360          // default, sets full circle
37
38 }
```

30.4 Publications and Subscriptions for `uFldContactRangeSensor`

The interface for `uFldContactRangeSensor`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldContactRangeSensor --interface or -i
```

30.4.1 Variables Published by uFldContactRangeSensor

The primary output of `uFldContactRangeSensor` to the MOOSDB is the posting of range reports and visual cues for the range reports.

- `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 30.5.
- `CRS_RANGE_REPORT`: A report on the range from a particular vehicle to the pinging vehicle. Section 30.2.4.
- `CRS_RANGE_REPORT_NAMEJ`: A report on the range from a particular named NAMEJ, to the pinging vehicle.
- `VIEW_RANGE_PULSE`: A description for visualizing the beacon range report. Section 30.6.

30.4.2 Variables Subscribed for by uFldContactRangeSensor

The `uFldContactRangeSensor` 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 11.10.4.
- `CRS_RANGE_REQUEST`: A request to generate range reports for all targets to all vehicles within range of the target.
- `NODE_REPORT`: A report on a vehicle location and status.
- `NODE_REPORT_LOCAL`: A report on a vehicle location and status.

Command Line Usage of uFldContactRangeSensor

The `uFldContactRangeSensor` 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:

```
$ uFldContactRangeSensor --help or -h
```

This will show the output shown in Listing 3 below.

Listing 3 - Command line usage for the uFldContactRangeSensor tool.

```
1  Usage: uFldContactRangeSensor file.moos [OPTIONS]
2
3  Options:
4      --alias=<ProcessName>
5          Launch uFldContactRangeSensor with the given process
6          name rather than uFldContactRangeSensor.
7      --example, -e
8          Display example MOOS configuration block
9      --help, -h
10     Display this help message.
```

```

11 --version,-v
12     Display the release version of uFldContactRangeSensor.

```

30.5 Terminal and AppCast Output

The `uFldContactRangeSensor` application produces some useful information to the terminal and identical content through appcasting. An example is shown in Listing 4 below. On line 2, the name of the local community or vehicle name is listed on the left. On the right, "0/0(848) indicates there are no configuration or run warnings, and the current iteration of `uFldContactRangeSensor` is 848. Lines 4-10 convey the prevailing configuration settings. They may reflect the default values or values read from the configuration block like the one shown previously in Listing 2.

Listing 4 - Example `uFldContactRangeSensor` console or appcast output.

```

1 =====
2 uFldContactRangeSensor shoreside          0/0(848)
3 =====
4 Default Ping Wait:      30
5 Random Noise Algorithm: uniform
6 Random Noise Uniform Pct: 0.04
7 Gaussian Noise:        0
8 Ping Color:            white
9 Echo Color:             chartreuse
10 Ground Truth Reporting: true
11 Allow Echo Types:      all types
12
13 Vehicles(3):
14      Node  Ping  Push  Pull  Pings  Echos  Too   Too   Echos
15 VName   Reps  Wait  Dist  Dist  Gen'd  Rec'd  Freq  Far   Sent
16 -----
17 archie  411   32    190   100   11     5      6     0     0
18 betty   410   30    100   100   0      0      0     0     0
19 jackal  411   30    50    100   0      0      0     0     5
20
21 =====
22 Most Recent Events (8):
23 =====
24 [211.60]: ARCHIE <-- JACKAL
25 [211.60]: ARCHIE ----))
26 [181.40]: ARCHIE ----)) XXX too frequent. (4.03)
27 [177.37]: ARCHIE <-- JACKAL
28 [177.37]: ARCHIE ----))
29 [146.23]: ARCHIE ----)) XXX too frequent. (30.66)
30 [144.22]: ARCHIE ----)) XXX too frequent. (28.64)
31 [115.58]: ARCHIE <-- JACKAL

```

In the block of output beginning on line 13, the information is organized by vehicle name in each row. The first two columns show the vehicle name and number of node reports received for that vehicles.

The next three columns reflect configuration settings for particular vehicles. For example, the *Ping Wait* time of 32 seconds reflects line 17 in Listing 2. The *Push Dist* value reflects line 16 in Listing 2. The last five columns reflect the results of pings generated by vehicles. THe *Pings Gen'd* column shows how many `CRS_SENSOR_REQUEST` messages have been received by that vehicle. The next column shows how many replies were sent back to the vehicle. The *Too Freq* and *Too Far* columns explain why a ping may not have been replied. The *Echos Sent* column shows how many echos were

sent to others about that vehicle. In this snapshot, archie has received five ping echos about the jackal vehicle.

The last block, lines 21-31 show the most recent events with their timestamps. Line 28 shows an example where a ping was generated by archie, followed by a reply about jackal on line 27. Lines like 26 indicate that a ping was rejected by the simulator as being too soon after the previous ping.

30.6 Interaction between uFldContactRangeSensor and pMarineViewer

The `uFldContactRangeSensor` application will post certain messages to the MOOSDB that may be subscribed for by GUI based applications like `pMarineViewer` for visualizing the posting of `CRS_RANGE_REPORT` and `CRS_RANGE_REQUEST` messages. A *range pulse* message conveys visually the generation of a range report, or the receipt of a range request. The pulse is rendered as a ring with a growing radius having the vehicle at the center. Range pulses are posted to the MOOS variable `VIEW_RANGE_PULSE` with a form similar to:

```
VIEW_RANGE_PULSE = "x=-40,y=-150,radius=40,duration=15,fill=0.25,fill_color=green,
label=04,edge_color=green,time=3892830128.5,edge_size=1"
```

The `x` and `y` parameters convey the center of the pulse. The `radius` parameter indicates the radius of the circle at its maximum. The `duration` parameter is the number of seconds the pulse will be rendered. The pulse will grow its radius linearly from zero meters at zero seconds to `radius` meters at `duration` seconds. The `fill` parameter is in the range [0, 1], where 0 is full transparency (clear) and 1 is fully opaque. The pulse transparency increases linearly as the range ring is rendered. The starting transparency at `radius` = 0 is given by the `fill` parameter. The transparency at the maximum radius is zero. The `fill_color` parameter specifies the color rendered to the internal part of the range pulse. The choice of legal colors is described in Appendix B. The `label` is a string that uniquely identifies the range instance to consumers like `pMarineViewer`. As with other objects in `pMarineViewer`, if it receives an object the same label and type as one previously received, it will replace the old object with the new one in its memory. The `edge_color` parameter describes the color of the ring rendered in the range pulse. The `edge_size` likewise describes the line width of the rendered ring. The `time` parameter indicates the UTC time at which the range pulse object was generated.

One further note to developers of other apps perhaps wishing to also generate a range pulse for viewing - the recommended manner to generate a range pulse string is to create an instance of the `XYRangePulse` class using the defined class interface. The string should be obtained by invoking the serialization method for that class. This will better ensure compatibility as the software evolves. The class is defined in `lib_geometry` in the MOOS-IvP tree.

30.7 The Hugo Example Mission Using uFldContactRangeSensor

The *hugo* mission is distributed with the MOOS-IvP source code and contains a ready example of the `uFldContactRangeSensor` application. Assuming the reader has downloaded the source code available at www.moos-ivp.org and built the code according to the discussion in Section 1.2.9, the *hugo* mission may be launched by:

```
$ cd moos-ivp/ivp/missions/m8_hugo/
$ ./launch.sh 10
```

The argument, 10, in the line above will launch the simulation in 10x real time. Once this launches, the [pMarineViewer](#) GUI application should launch and the mission may be initiated by hitting the DEPLOY button. Two vehicles should be visibly moving, one surface vehicle labeled "archie" and one UUV labeled "jackal", as shown in Figure 113.

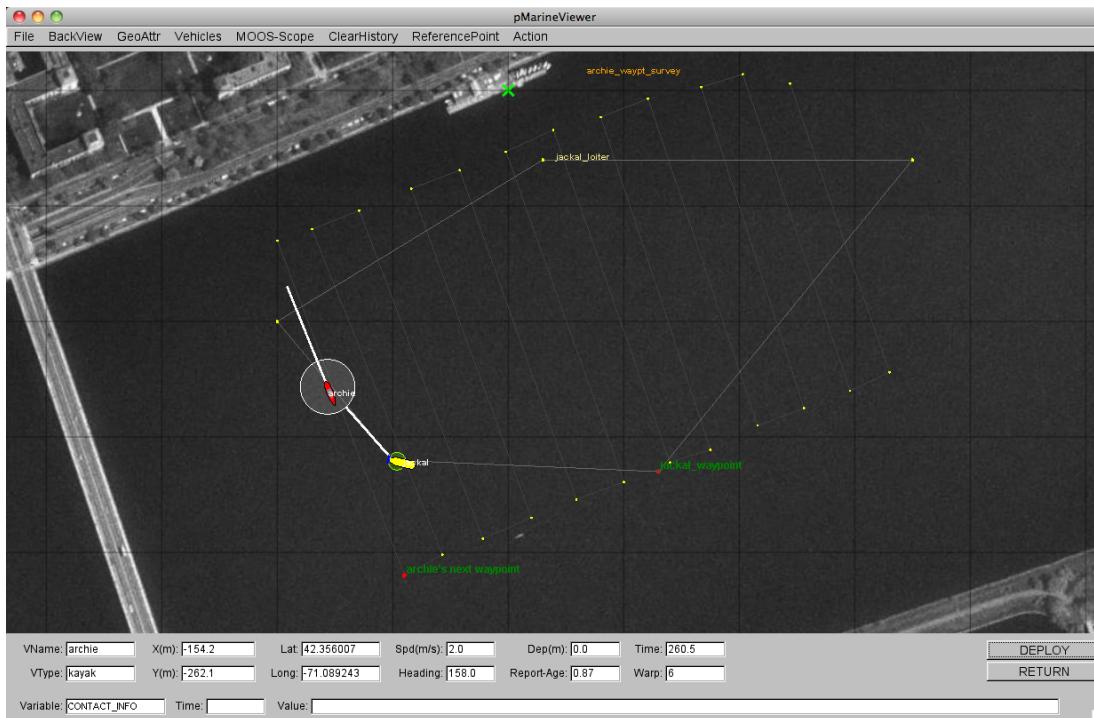


Figure 113: The Hugo Example Mission: The Hugo example mission involves two simulated vehicles. The first vehicle is a surface vehicle, *archie*, traversing a lawnmower shaped search pattern. The second vehicle, is a UUV, *jackal*, traversing a polygon pattern overlapping the first pattern. Periodically *archie* emits a ping (range request). This example contains three distinct MOOS communities - one for each simulated vehicle, and one for the shoreside community running `uFldContactRangeSensor`.

The surface vehicle will traverse a survey pattern over the region shown in Figure 113, periodically generating a range request (ping). With each range request, a white range pulse should be visible around the vehicle. Almost immediately afterwards, a range report for each neighboring vehicle within range is generated and a green range pulse around each “target” vehicle is rendered. The snapshot in Figure 113 depicts a moment in time where the range request visual pulses are around both vehicles. The larger pulse around the surface vehicle indicates it was generated just prior to the reply pulse around the UUV.

How does the simulated vehicle generate a range request? In practice a user may implement an intelligent module to reason about when to generate requests, but in this case the [uTimerScript](#) application is used by creating a script that repeats endlessly, generating a range request once

every 25-35 seconds. The script is also conditioned on (`NAV_SPEED > 0`), so the pinging doesn't start until the vehicle is deployed. The configuration for the script can be seen in the `uTimerScript` configuration block in the `shoreside.moos` file. More on the `uTimerScript` application can be found in the `uTimerScript` section in [2].

31 uFldCollisionDetect: Detecting Collisions

31.1 Overview

The `uFldCollisionDetect` application is run by the shoreside and detects if any two vehicles have collided. If a collision is detected, then various actions are possible including: 1) publishing the collision data to a single variable, 2) publishing the collision data to a variable specific to the two colliding vehicles, or 3) displaying a range pulse at the site of the collision in pMarineViewer. Collision data also appears as an AppCast display within pMarineViewer.

31.2 Using uFldCollisionDetect

All vehicles publishing to `NODE_REPORT` will be subscribed to `uFldCollisionDetect`. There is no need to implement the `uFldCollisionDetect` application anywhere other than the shoreside.moos file.

31.2.1 Configuring the Collision Distance

A collision is detected and specified action is taken whenever the range between any two vehicles is less than the specified collision range. The collision range is defined by the user using the `COLLISION_RANGE` configuration parameter in the shoreside.moos file. The default value is 5 [meters]. Distances between vehicles are measured using X & Y location values published in `NODE_REPORT`. If multiple vehicles are within a collision range of another vehicle at the same time, then all collision cases will be reported and action will be taken for all collision cases simultaneously.

31.2.2 AppCasting with uFldCollisionDetect

The AppCast displays the detection range and minimum range achieved of each vehicle pair that is colliding. The minimum range is the minimum range as seen from the current onset of a collision condition until the collision condition clears; it is not the global minimum range of all interactions between these two vehicles. The display of collision information appears immediately upon a collision detection and clears when the two vehicles are no longer within the specified collision range of each other. The clearing of the collision information from the AppCast display can be delayed from clearing using the `DELAY_TIME_TO_CLEAR` parameter to allow the user sufficient time to view the information. The default value for `DELAY_TIME_TO_CLEAR` is 5 [seconds].

```

=====
uFldCollisionDetect shoreside          0/0 (4716)
=====
=====

uFldCollisionDetect
=====

    Threshold Collision Range:    10
    Delay Time to Clear Messages: 35
    Post to MOOSDB Immediately:   true

Vehicle 1  Vehicle 2  Min Distance  Detection Distance
-----  -----  -----
archie      betty       3.49857      9.74559
betty      charlie     2.00340      7.87391
=====

Most Recent Events (8):
=====
[1156.68]: new collision detected: v1=archie,v2=betty,detection_distance=9.75
[1151.67]: new collision detected: v1=betty,v2=charlie,detection_distance=7.87
[1114.04]: new collision detected: v1=archie,v2=charlie,detection_distance=9.82
[1068.38]: new collision detected: v1=betty,v2=charlie,detection_distance=9.74
[1056.28]: new collision detected: v1=archie,v2=betty,detection_distance=8.82
[986.76]: new collision detected: v1=betty,v2=charlie,detection_distance=8.68
[955.60]: new collision detected: v1=archie,v2=betty,detection_distance=9.15
[904.43]: new collision detected: v1=betty,v2=charlie,detection_distance=9.50

```

Figure 114: **AppCasting with uFldCollisionDetect:** Two vehicle pairs are within collision range of each other. Betty is considered to be colliding with both archie and charlie; archie and charlie are not colliding. The minimum distance is the closest that each pair has been during this collision. A list of previous collisions is kept in the most recent events buffer.

31.2.3 Range Pulses with uFldCollisionDetect

Range pulses can be configured to show the user a visual signal that a collision has occurred. The range pulse can be configured to change both its range and duration using configuration parameters.

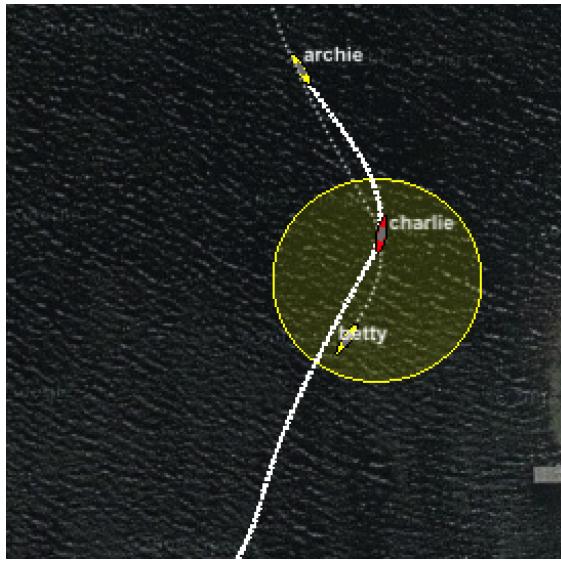


Figure 115: **Using Range Pulse with uFldCollisionDetect:** A range pulse is displayed in pMarineViewer with each detected collision.

31.3 Configuration Parameters of uFldCollisionDetect

The following parameters are defined for `uFldCollisionDetect`. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so.

Listing 31.1: Configuration parameters for `uFldCollisionDetect`.

- `collision_range`: The range in meters that is considered a collision. Default value is 5 meters.
- `delay_time_to_clear`: The time delay from a pair of vehicles exiting a collision condition until the AppCast information is cleared.
- `deploy_delay_time`: The time delay from the deploy command being issued until the first collision will be registered. This allows for vehicles to be deployed inside the specified collision range and given the specified amount of time to open range before collision reports are generated.
- `pulse`: When set to TRUE, a range pulse will be used at the location of each collision to visually identify the situation in pMarineViewer.
- `pulse_range`: Sets the range of collision range pulses. Default value is 20 meters.
- `pulse_duration`: Sets the duration of collision range pulses. Default value is 4 seconds.

- `publish_immediately`: When `publish_immediately = true`, the value of "false" is published immediately to the appropriate variable (depending on values of `publish_single` and `publish_pairs`) upon clearing of the collision condition. When `publish_immediately = false`, the value of "false" is published to the appropriate variable after `delay_time_to_clear` seconds have elapsed since the collision condition cleared.
- `publish_pairs`: If set to `true`, a parameter specific to each vehicle pair will be published for each collision. If set to `false`, this vehicle-pair-specific variable will not be published. Both `publish_pairs` and `publish_single` can be set to `true` simultaneously if desired. This takes the form `vehicle_collision_${v1}_${v2} = detection_distance`, where `$v1` and `$v2` are names of the colliding vehicles.
- `publish_single`: if set to `true`, a single variable will be used for all collisions specifying the two colliding vehicles within the published string. This takes the form `vehicle_collision="v1=archie,v2=betty,detection_distance=4.9"`.

An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command line:

```
$ uFldCollisionDetect --example or -e
```

This will show the output shown in Listing 2 below.

Listing 31.2: Example configuration of the `uFldCollisionDetect` application.

```

1 =====
2 uCollisionDetector Example MOOS Configuration
3 =====
4
5 ProcessConfig = uFldCollisionDetect
6 {
7   AppTick    = 4
8   CommsTick = 4
9   COLLISION_RANGE = 10 // range considered a collision
10  DELAY_TIME_TO_CLEAR = 35
11    // delay time [seconds] from collision condition clearing to
12    removal from AppCasting
13  DEPLOY_DELAY_TIME = 10 // default is 0 seconds
14  PUBLISH_IMMEDIATELY = true
15    // publish the cleared variable to MOOSDB immediately after
16    collision condition clears.
17  PULSE = TRUE
18  PULSE_RANGE = 50
19  PULSE_DURATION = 15
20
21  PUBLISH_PAIRS = FALSE //DEFAULT FALSE
22  PUBLISH_SINGLE = TRUE //DEFAULT TRUE
23 }
```

31.4 Publications and Subscriptions for uFldCollisionDetect

The interface for `uFldCollisionDetect`, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ uFldCollisionDetect --interface or -i
```

31.4.1 Variables Published by uFldCollisionDetect

The primary output of `uFldCollisionDetect` to the MOOSDB is the posting of collision reports and visual cues for the collision reports. The `uFldCollisionDetect` application is capable of publishing variables specific to the vehicles involved in the collision and / or to a generic collision variable depending on configuration parameters. For users wanting to analyze only interactions between two specific vehicles, publishing of the `VEHICLE_COLLISION_${V1}_${V2}` parameter may be preferable to publishing all collision data then later sifting through log files.

31.4.2 Vehicle-Specific Publication

A variable `VEHICLE_COLLISION_${V1}_${V2}` is published for each collision with the initial detection distance. This is published once per collision. An update is published with "false" when a collision condition clears. This publication can be delayed to correspond with the clearing of the AppCast display using the parameter `PUBLISH_IMMEDIATELY`. When `PUBLISH_IMMEDIATELY = TRUE`, the value of "false" is published immediately upon clearing of the collision condition. When the value is FALSE, the value of "false" is published when the `DELAY_TIME_TO_CLEAR` time has elapsed.

To use vehicle-specific publication, set the variable `PUBLISH_PAIRS = TRUE`.

31.4.3 Generic Collision Publication

A variable `VEHICLE_COLLISION` is published for each collision with the initial detection distance. This is published once per collision. An update is published with "false" when a collision condition clears. This publication can be delayed to correspond with the clearing of the AppCast display using the parameter `PUBLISH_IMMEDIATELY`. When `PUBLISH_IMMEDIATELY = TRUE`, the value of "false" is published immediately upon clearing of the collision condition. When the value is FALSE, the value of "false" is published when the `DELAY_TIME_TO_CLEAR` time has elapsed.

To use non-specific collision publication, set the variable `PUBLISH_SINGLE = TRUE`.

```
VEHICLE_COLLISION_${V1}_${V2} = detection_distance  
(when new collision detected)  
(published only once per collision)
```

```
VEHICLE_COLLISION_${V1}_${V2} = false  
(when existing collision clears)  
(published only once per collision)
```

where \$V1 and \$V2 are vehicle names.

```
VEHICLE_COLLISION="v1=archie,v2=betty,detection_distance=4.9"
```

31.4.4 Variables Subscribed for by uFldCollisionDetect

The `uFldCollisionDetect` subscribes to `NODE_REPORT` and `DEPLOY_ALL`. Each vehicle that publishes a node report will be analyzed by `uFldCollisionDetect`. When vehicles are deployed, ranges are analyzed for collision conditions subject to the `DEPLOY_DELAY_TIME` parameter.

```
NODE_REPORT = NAME=alpha,TYPE=UUV,TIME=1252348077.59,  
X=51.71,Y=-35.50,LAT=43.824981,  
LON=-70.329755,SPD=2.0,HDG=118.8,  
YAW=118.8,DEPTH=4.6,LENGTH=3.8,  
MODE=MODE@ACTIVE:LOITERING  
  
DEPLOY_ALL = TRUE
```

Command Line Usage of uFldCollisionDetect

The `uFldCollisionDetect` 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:

```
$ uFldCollisionDetect --help or -h
```

This will show the output shown in Listing 3 below.

Listing 31.3: Command line usage for the `uFldCollisionDetect` tool.

```
1 Usage: uFldCollisionDetect file.moos [OPTIONS]  
2  
3 Options:  
4   --alias=<ProcessName>  
5       Launch uFldCollisionDetect with the given process  
6       name rather than uFldCollisionDetect.  
7   --example, -e  
8       Display example MOOS configuration block  
9   --help, -h  
10      Display this help message.  
11   --version,-v  
12      Display the release version of uFldCollisionDetect.
```

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 116:

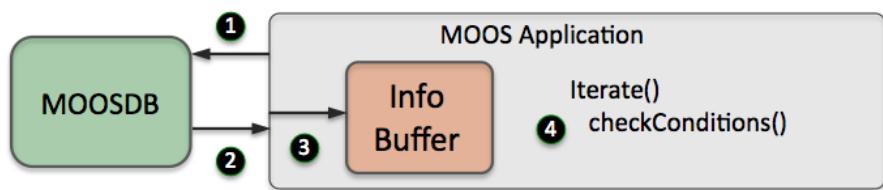


Figure 116: **Logic conditions in a MOOS application:** Step 1: the applications registers to the MOOSDB for any MOOS variables involved in the logic expressions. Step 2: The MOOS application reads incoming mail from the MOOSDB. Step 3: Any new mail results in an update to the information buffer. Step 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 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 delimiters. Mixed delimiters 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)	firebrick (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)
lightblue (ad,d8,e6)
lightcyan (e0,ff,ff)
lightgray (d3,d3,d3)
lightpink (ff,b6,c1)
lightseagreen (20,b2,aa)
lightslategray (77,88,99)
lightyellow (ff,ff,e0)
limegreen (32,cd,32)
magenta (ff,00,ff)
mediumblue (00,00,cd)
mediumseagreen (3c,b3,71)
mediumspringgreen (00,fa,9a)
mediumvioletred (c7,15,85)
mintcream (f5,ff,fa)
moccasin (ff,e4,b5)
navy (00,00,80)
olive (80,80,00)
orange (ff,a5,00)
orchid (da,70,d6)
paleturquoise (af,ee,ee)
papayawhip (ff,ef,d5)
pelegoldenrod (ee,e8,aa)
pink (ff,c0,cb)
powderblue (b0,e0,e6)
red (ff,00,00)
royalblue (41,69,e1)
salmon (fa,80,72)
seagreen (2e,8b,57)
sienna (a0,52,2d)
skyblue (87,ce,eb)
slategray (70,80,90)
springgreen (00,ff,7f)
tan (d2,b4,8c)
thistle (d8,bf,d8)
turquoise (40,e0,d0)
wheat (f5,de,b3)
whitesmoke (f5,f5,f5)
yellowgreen (9a,cd,32)
lemonchiffon (ff,fa,cd)
lightcoral (f0,80,80)
lightgoldenrod (fa,fa,d2)
lightgreen (90,ee,90)
lightsalmon (ff,a0,7a)
lightskyblue (87,ce,fa)
lightsteelblue (b0,c4,de)
lime (00,ff,00)
linen (fa,f0,e6)
maroon (80,00,00)
mediumorchid (ba,55,d3)
mediumslateblue (7b,68,ee)
mediumturquoise (48,d1,cc)
midnightblue (19,19,70)
mistyrose (ff,e4,e1)
navajowhite (ff,de,ad)
oldlace (fd,f5,e6)
olivedrab (6b,8e,23)
orangered (ff,45,00)
palegreen (98,fb,98)
palevioletred (db,70,93)
peachpuff (ff,da,b9)
peru (cd,85,3f)
plum (dd,a0,dd)
purple (80,00,80)
rosybrown (bc,8f,8f)
saddlebrown (8b,45,13)
sandybrown (f4,a4,60)
seashell (ff,f5,ee)
silver (c0,c0,c0)
slateblue (6a,5a,cd)
snow (ff,fa,fa)
steelblue (46,82,b4)
teal (00,80,80)
tomatao (ff,63,47)
violet (ee,82,ee)
white (ff,ff,ff)
yellow (ff,ff,00)

References

- [1] 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.
- [2] Michael R. Benjamin. MOOS-IvP Autonomy Tools Users Manual Release 4.2.1. Technical Report MIT-CSAIL-TR-201136, MIT Computer Science and Artificial Intelligence Lab, July 2011.
- [3] Michael R. Benjamin, Paul M. Newman, Henrik Schmidt, and John J. Leonard. An Overview of MOOS-IvP and a Brief Users Guide to the IvP Helm Autonomy Software. Technical Report MIT-CSAIL-TR-2009-028, MIT Computer Science and Artificial Intelligence Lab, June 2009.
- [4] Mary M. Hunt, William M. Marquet, Donald A. Moller, Kenneth R. Peal, Woollcott K. Smith, and Rober C. Spindel. An Acoustic Navigation System. Technical Report WHOI-74-6, Woods Hole Oceanographic Institution, Woods Hole, Massachusetts, December 1974.
- [5] P. A. Milne. *Underwater Acoustic Positioning Systems*. Gulf Publishing Co., Houston TX, January 1983.
- [6] Roger P. Sokey and Thomas C. Austin. Sequential Long-Baseline Navigation for REMUS, an Autonomous Underwater Vehicle. *Proceedings of SPIE, the International Society for Optical Engineering*, 3711:212–219a, 1999.
- [7] Louis L. Whitcomb, Dana R. Yoerger, and Hanumant Singh. Combined Doppler/LBL Based Navigation of Underwater Vehicles. In *11th International Symposium on Unmanned Untethered Submersible Technology (UUST99)*, Durham, New Hampshire, August 1999.

Index

- alogclip, 19
 - Command Line Usage, 206
 - Example Output, 207
 - Overview, 206
- aloggrep, 19
 - Command Line Usage, 208
 - Example Output, 208
 - Overview, 207
- alogrm, 19
 - Command Line Usage, 209
 - Example Output, 210
 - Overview, 209
- alogscan, 19
 - Command Line Usage, 203
 - Example Output, 204
 - Overview, 203
- alogview, 19
 - Command Line Usage, 212
 - Overview, 211
 - Vehicle Trajectories, 213
- AppCasting, 26
 - pMarineViewer, 53
- AppCastingMOOSApp::Iterate(), 167
- AppCastingMOOSApp::OnNewMail(), 165
- AppCastingMOOSApp::RegisterVariables(), 166
- Behavior-Posts, 71
- buildReport(), 164–167
- Buoyancy, 150
- Command and Control
 - uPokeDB, 182
 - uTermCommand, 198
- Command Line Usage
 - alogclip, 206
 - aloggrep, 208
 - alogrm, 209
 - alogscan, 203
 - alogview, 212
- pBasicContactMgr, 131
- pHostInfo, 179
- pNodeReporter, 84
- pSearchGrid, 195, 197
- uFldBeaconRangeSensor, 309
- uFldCollisionDetect, 338
- uFldContactRangeSensor, 329
- uFldHazardMetric, 295
- uFldHazardMgr, 285
- uFldHazardSensor, 275
- uFldMessageHandler, 243
- uFldNodeBroker, 218
- uFldNodeComms, 238
- uFldPathCheck, 253
- uFldScope, 248
- uFldShoreBroker, 229
- uHelmScope, 74
- uMACView, 158
- uPokeDB, 182
- uProcessWatch, 139
- uSimCurrent, 201

uSimMarine, 142, 145
 uTermCommand, 198
 uTimerScript, 113
 uXMS, 94, 99
 Contact Management, 121
 Depth Simulation, 150
 Example Config Block
 uTimerScript, 115
 GPS, 116
 handleConfigFOO(value), 171
 Iterate(), 164, 165
 IvP Behavior Parameters
 updates, 70
 IvP Behaviors
 Dynamic Configuration, 70
 IvP Domain, 69
 varbalk, 69
 IvP Helm
 Contact Management, 127
 Logic Expressions, 342
 MOOS
 Acronym, 21
 Background, 20
 Documentation, 23
 Operating Systems, 23
 Source Code, 21
 Sponsors, 21
 Mouse
 Mouse Poke Configuration, 58
 Mouse Pokes in pMarineViewer, 58
 OnConnectToServer(), 166
 OnNewMail(), 164, 165
 OnStartUp(), 164, 166, 170
 pBasicContactMgr, 18, 121
 Command Line Usage, 131
 Configuration Parameters, 128
 Coordination with the IvP Helm, 127
 Publications and Subscriptions, 130
 pEchoVar, 18, 186
 Configuration Parameters, 186, 189
 Configuring Echo Event Mappings, 186
 Publications and Subscriptions, 190
 Variable Flipping, 186
 pHHostInfo, 19, 177
 Command Line Usage, 179
 Configuration Parameters, 178, 179
 Publications and Subscriptions, 178
 pMarineViewer, 17, 33
 Actions, 56
 Background Images, 40
 Configuration Parameters, 61
 Drop Points, 49
 Full-Screen Mode, 42
 Geometric Objects, 43
 Hash Marks, 42
 Image Shifting To Vehicles, 52
 Markers, 45
 Panning and Zooming, 39
 Poking the MOOSDB, 56, 58
 Publications and Subscriptions, 66
 Pull-Down Menu (Action), 56
 Pull-Down Menu (AppCasting), 53
 Pull-Down Menu (BackView), 39
 Pull-Down Menu (GeoAttributes), 42
 Pull-Down Menu (MouseContext), 58
 Pull-Down Menu (ReferencePoint), 59
 Pull-Down Menu (Scope), 56
 Pull-Down Menu (Vehicles), 50
 Range Pulses, 46, 48
 Stale Vehicles, 51
 Vehicle Colors, 52
 Vehicle Name Mode, 50
 Vehicle Shapes, 51
 Vehicle Trails, 52
 pNodeReporter, 18, 77, 86
 Command Line Usage, 84
 Configuration Parameters, 82, 84, 86
 Publications and Subscriptions, 83
 pSearchGrid, 18, 193
 Command Line Usage, 197
 Configuration Parameters, 195, 197
 Publications and Subscriptions, 196
 Publications
 uFldCollisionDetect, 340

uFldContactRangeSensor, 331
 Publications and Subscriptions
 pBasicContactMgr, 130
 pEchoVar, 190
 pHostInfo, 178
 pMarineViewer, 66
 pNodeReporter, 83
 pSearchGrid, 196
 uFldBeaconRangeSensor, 311
 uFldCollisionDetect, 340
 uFldContactRangeSensor, 330
 uFldHazardMetric, 296
 uFldHazardMgr, 288
 uFldHazardSensor, 277
 uFldMessageHandler, 243
 uFldNodeBroker, 219
 uFldNodeComms, 240
 uFldPathCheck, 253
 uFldScope, 248
 uFldShoreBroker, 230
 uHelmScope, 75
 uPokeDB, 184
 uProcessWatch, 140
 uSimCurrent, 201
 uSimMarine, 144
 uTermCommand, 200
 uTimerScript, 114
 uXMS, 103

 Range Pulses
 pMarineViewer, 48
 registerVariables(), 166
 reportConfigWarning(), 169, 171
 reportEvent(), 168
 reportRunWarning(), 168
 reportUnhandledConfigWarning(), 171

 Scoping the MOOSDB
 uHelmScope, 70
 Source Code
 Building, 21
 Obtaining, 21
 Stale Vehicles
 pMarineViewer, 51
 Staleness
 uFldNodeComms, 233

 Start Delay
 uTimerScript, 111

 Thrust Map, 154
 Time Warp
 uTimerScript, 110

 uFldBeaconRangeSensor, 20, 308
 Command Line Usage, 312
 Configuration Parameters, 309
 Publications and Subscriptions, 311
 uFldCollisionDetect, 336
 Command Line Usage, 341
 Configuration Parameters, 338
 Publications, 340
 Publications and Subscriptions, 340
 uFldContactRangeSensor, 20, 323
 Command Line Usage, 331
 Configuration Parameters, 329
 Publications, 331
 Publications and Subscriptions, 330
 uFldHazardMetric, 20, 291
 Command Line Usage, 297
 Configuration Parameters, 295
 Publications and Subscriptions, 296
 uFldHazardMgr, 20, 283
 Command Line Usage, 289
 Configuration Parameters, 285
 Publications and Subscriptions, 288
 uFldHazardSensor, 20, 255
 Command Line Usage, 278
 Configuration Parameters, 275
 Publications and Subscriptions, 277
 uFldMessageHandler, 20, 242
 Command Line Usage, 244
 Configuration Parameters, 243
 Publications and Subscriptions, 243
 uFldNodeBroker, 19, 217
 Command Line Usage, 220
 Configuration Parameters, 218
 Publications and Subscriptions, 219
 uFldNodeComms, 20, 232
 Asymmetric Intervehicle Message Arrival, 234
 Configuration Parameters, 238
 CriticalRange, 233

Enhanced Stealth, 234
 Handling Node Reports, 233
 Maximum Message Length, 236
 Message Handling, 235
 Minimum Message Length, 236
 Publications and Subscriptions, 240
 Routing Criteria, 235
 Staleness, 233
 Transmissin with pShare, 235
uFldPathCheck, 20, 252
 Configuration Parameters, 253
 Publications and Subscriptions, 253
uFldScope, 20, 247
 Command Line Usage, 249
 Configuration Parameters, 248
 Publications and Subscriptions, 248
uFldShoreBroker, 19, 223
 Command Line Usage, 230
 Configuration Parameters, 229
 Publications and Subscriptions, 230
uHelmScope, 17, 68
 Configuration Parameters, 74
 IvP Domain, 69
 Publications and Subscriptions, 75
 Scoping the MOOSDB, 70
 User Input, 71
uMAC, 17
uMACView, 18
 Configuration Parameters, 158
uPokeDB, 18, 182
 Command Line Usage, 182
 Publications and Subscriptions, 184
uProcessWatch, 17, 133
 Configuration Parameters, 139
 Publications and Subscriptions, 140
uSimCurrent, 18, 201
 Configuration Parameters, 201
 Publications and Subscriptions, 201
uSimMarine, 18, 118, 142
 Command Line Usage, 145
 Configuration Parameters, 142, 145
 Depth Simulation, 150
 Initial Vehicle Position and Pose, 146
 Propagating Vehicle Altitude, 151
 Propagating Vehicle Position and Pose, 146
 Publications and Subscriptions, 144
 Resetting, 146
 Thrust Map, 154
uTermCommand, 19, 198
 Command and Control, 198
 Configuration Parameters, 198
 Publications and Subscriptions, 200
uTimerScript, 19, 105, 321, 334
 Arithmetic Expressions, 110
 Atomic Scripts, 108
 Conditional Pausing, 108
 Configuration Parameters, 108, 113
 Configuring the Event List, 106
 Example Config Block, 115
 Exiting, 108
 Fast Forwarding in Time, 108
 Jumping To the Next Event, 108
 Logic Conditions, 108
 Macros, 109, 118, 120
 Macros Built-In, 109
 Pausing the Script, 107
 Pausing the Script with Conditions, 108
 Publications and Subscriptions, 114
 Quitting, 108
 Random Variables, 109
 Resetting, 106, 118, 120
 Restart Delays, 111
 Script Flow Control, 107, 108
 Simulated GPS Unit, 116
 Simulated Random Wind Gusts, 118
 Simulated Range Requests, 321, 334
 Start Delay, 111, 118, 120
 Start Delays, 111
 Status Messages, 111
 Time Warp, 110
uXMS, 17, 70, 88
 Command Line Usage, 99
 Configuration Parameters, 94, 99
 Console Interaction, 101
 Publications and Subscriptions, 103
varbalk
uHelmScope, 69

Virgin Variables, [71](#)

Wind, [118](#)

YourMOOSApp::OnNewMail(), [165](#)

YourMOOSApp::OnStartUp(), [166](#)

Index of MOOS Variables

DESIRED_HEADING, 91
MVIEWER_LCLICK, 58
MVIEWER_RCLICK, 58
PROC_WATCH_EVENT, 133
PROC_WATCH_SUMMARY, 133
TRAIL_RESET, 53
APPCAST_REQ_<COMMUNITY>, 66
APPCAST_REQ, 54, 66, 67, 75, 84, 104, 115, 131, 140, 145, 158, 160, 162, 174, 179, 190, 197, 219, 230, 241, 244, 249, 253, 278, 288, 296, 312, 331
APPCAST, 66, 67, 75, 84, 104, 115, 130, 140, 144, 158, 160–162, 179, 190, 196, 219, 230, 240, 244, 249, 253, 277, 288, 296, 311, 331
BCM_ALERT_REQUEST, 123, 131
BCM_DISPLAY_RADII, 123, 131
BCM_OK, 136
BHV_WARNING, 69
BRS_RANGE_REPORT_GT, 317
BRS_RANGE_REPORT_NAMEJ, 311
BRS_RANGE_REPORT, 308, 311, 314, 316, 317, 319, 322
BRS_RANGE_REQUEST, 308, 312, 314, 319
BUOYANCY_RATE, 151
BUOYANCY_REPORT, 144
CONTACTS_ALERTED, 125, 130
CONTACTS_LIST, 125, 130
CONTACTS_RECAP, 126, 130
CONTACTS_RETired, 126, 130
CONTACTS_UNALERTED, 126, 130
CONTACT_MGR_WARNING, 130
CONTACT_RESOLVED, 126, 131
CRS_RANGE_REPORT_GT, 329
CRS_RANGE_REPORT_NAMEJ, 331
CRS_RANGE_REPORT, 323, 329, 331, 333
CRS_RANGE_REQUEST, 323–326, 331, 333
CRS_SENSOR_REQUEST, 332
DB_CLIENTS, 17, 93, 102, 104, 133, 135, 136, 138–140
DB_UPTIME, 104, 109
DEPLOY_ALL, 341
DEPLOY, 306
DESIRED_ELEVATOR, 142, 145, 146, 150
DESIRED_HEADING, 91, 93, 352
DESIRED_RUDDER, 142, 145–148
DESIRED_THRUST, 142, 145–148
DRIFT_VECTOR_ADD, 145, 152, 153
DRIFT_VECTOR_MULT, 145, 152
DRIFT_VECTOR, 145, 152, 201, 202
DRIFT_X, 145, 152, 201
DRIFT_Y, 145, 152, 201
EXITED_NORMALLY, 109, 115, 137, 141
GPS RECEIVED, 117
GPS_UPDATE RECEIVED, 117, 118
HAZARDSET_EVAL_<VNAME>, 296
HAZARDSET_EVAL_FULL, 295, 296
HAZARDSET_EVAL, 294, 296, 302
HAZARDSET_REPORT_EVAL, 292
HAZARDSET_REPORT, 284, 285, 288, 292, 293, 297, 302, 303
HAZARDSET_REQUEST=true, 305
HAZARDSET_REQUEST, 284, 288, 299, 301, 303
HAZARD_SEARCH_SCORE, 296
HAZARD_SEARCH_START, 294, 297, 298, 302
HELM_MAP_CLEAR, 66
HOST_INFO_REQUEST, 179
IVPHELM_ALLSTOP, 84
IVPHELM_DOMAIN, 74, 76
IVPHELM_LIFE_EVENT, 73, 76
IVPHELM_MODESET, 74, 76
IVPHELM_REJOURNAL, 73, 75
IVPHELM_STATEVARS, 70, 74, 76
IVPHELM_STATE, 73, 76, 78, 79, 84
IVPHELM_SUMMARY, 73, 76, 78, 79, 84
IVP_DOMAIN, 74
MAX_ACCELERATION, 147
MAX_DECELERATION, 147
MVIEWER_LCLICK, 58, 59, 66, 352
MVIEWER_RCLICK, 58, 59, 67, 352
NAV_DEPTH, 84, 117, 118, 142
NAV_GT_X, 80

NAV_GT_Y, 80
 NAV_HEADING, 84, 118, 131, 142
 NAV_LAT, 84
 NAV_LONG, 84
 NAV_SPEED, 78, 84, 118, 131, 142, 147
 NAV_X, 80, 84, 117, 118, 131, 142, 202
 NAV_Y, 84
 NODE_BROKER_ACK, 217, 219, 221, 224, 230
 NODE_BROKER_PING, 217, 219, 223, 228, 230
 NODE_MESSAGE_<VNAME>, 240
 NODE_MESSAGE, 233, 235, 241, 242, 244
 NODE_REPORT_<VNAME>, 240
 NODE_REPORT_LOCAL, 61, 67, 77, 78, 80, 82, 84,
 197, 214, 215, 241, 253, 278, 312, 331
 NODE_REPORT, 34, 67, 77, 78, 121, 131, 197, 233,
 235, 241, 247, 250, 253, 260, 278, 302,
 312, 318, 325, 331, 336, 341
 PHI_HOST_INFO, 67, 219, 221, 230
 PHI_HOST_IP_ALL, 179
 PHI_HOST_IP_VERBOSE, 179
 PHI_HOST_IP, 177, 179, 180
 PHI_HOST_PORT_DB, 179
 PHI_HOST_PORT_INFO, 179
 PLATFORM_REPORT_LOCAL, 82, 84, 85
 PLATFORM_REPORT, 77, 84
 PLAT_REPORT_SUMMARY, 85
 PLOGGER_CMD, 42, 67
 PMB_REGISTER, 230
 PNR_OK, 140
 POLYGON, 307
 PROC_WATCH_EVENT, 134, 136, 137, 140, 352
 PROC_WATCH_FULL_SUMMARY, 136, 137, 140
 PROC_WATCH_SUMMARY, 17, 93, 104, 133, 134, 136–
 140, 352
 PROC_WATCH_SUMMRY, 134
 PSG_GRID_RESET, 195, 197
 PSHARE_CMD, 219, 221
 PSHARE_INPUT_SUMMARY, 179
 RANGE_REPORT, 323
 RANGE_REQUEST, 316
 RETURN, 306
 ROTATE_SPEED, 145, 149
 SYSTEM_LENGTH, 79
 TRAIL_RESET, 53, 67, 352
 TRIM_REPORT, 144
 UHZ_CLASSIFY_REQUEST, 259, 302
 UHZ_CONFIG_ACK_ARCHE, 266
 UHZ_CONFIG_ACK, 259, 277, 284, 289, 302, 303
 UHZ_CONFIG_REQUEST, 259, 263, 265, 284, 286,
 288, 302, 303
 UHZ_DETECTION_REPORT_NAMEJ, 277
 UHZ_DETECTION_REPORT, 255, 259, 277, 284, 287,
 288, 302, 303
 UHZ_HAZARD_REPORT_NAMEJ, 277
 UHZ_HAZARD_REPORT, 255, 259, 277, 302
 UHZ_MISSION_PARAMS, 302
 UHZ_OPTIONS_SUMMARY, 266, 277, 302, 303
 UHZ_SENSOR_CONFIG, 278
 UHZ_SENSOR_REQUEST, 259, 278, 284, 287, 288,
 302, 303
 UMH_SUMMARY_MSGS, 244
 UNC_EARANGE, 235, 241
 UNC_STEALTH, 234, 241
 UNC_VIEW_NODE_RPT_PULSES, 241
 UPC_ODOMETRY_REPORT, 252, 253
 UPC_SPEED_REPORT, 252, 253
 UPC_TRIP_RESET, 253
 USC_CFIELD_SUMMARY, 202
 USER-DEFINED, 104
 USM_ALTITUDE, 144, 151
 USM_BUOYANCY_RATE, 145
 USM_CURRENT_FIELD, 145
 USM_DEPTH, 144, 145
 USM_DRIFT_SUMMARY, 145
 USM_DRIFT_VECTOR_ADD, 118
 USM_DRIFT_VECTOR_MULT, 152
 USM_DRIFT_VECTOR, 118
 USM_HEADING_OVER_GROUND, 145
 USM_HEADING, 145, 149
 USM_LAT, 145
 USM_LONG, 145
 USM_RESET_COUNT, 145
 USM_RESET, 145, 146
 USM_SIM_PAUSED, 145
 USM_SPEED_OVER_GROUND, 145
 USM_SPEED, 145, 148
 USM_WATER_DEPTH, 151
 USM_X, 145
 USM_YAW, 145

`USM_Y`, 145
`UTS_FORWARD`, 108, 114, 115
`UTS_NEXT`, 115
`UTS_PAUSE`, 107, 108, 114, 115
`UTS_RESET`, 106, 114, 115
`UTS_STATUS`, 111, 112, 114, 115
`VIEW_CIRCLE`, 67, 122, 130, 273, 277, 302
`VIEW_COMMs_PULSE`, 46, 67, 236, 237, 240
`VIEW_GRID_CONFIG`, 67
`VIEW_GRID_DELTA`, 67
`VIEW_GRID`, 67, 195, 196
`VIEW_MARKER`, 46, 63, 67, 215, 273, 277, 302, 311,
 314, 319, 321
`VIEW_POINT`, 67, 215
`VIEW_POLYGON`, 67, 215, 272, 277, 302
`VIEW_RANGE_PULSE`, 48, 67, 215, 311, 320, 322,
 324, 331, 333
`VIEW_SEGLIST`, 44, 67, 215
`VIEW_VECTOR`, 67, 202
`WATER_DEPTH`, 145
`hazard_file`, 257
`sensor_config`, 257

Index of Configuration Parameters: All MOOS Apps

AppTick, 189
COLLISION_RANGE, 336
DELAY_TIME_TO_CLEAR, 336, 340
DEPLOY_DELAY_TIME, 341
LatOrigin, 127
LongOrigin, 127
MOOSTimeWarp, 175, 225
PUBLISH_IMMEDIATELY = TRUE, 340
PUBLISH_IMMEDIATELY, 340
PUBLISH_PAIRS = TRUE, 340
PUBLISH_SINGLE = TRUE, 340
VEHICLE_COLLISION_\$V1_\$V2, 340
VEHICLE_COLLISION, 340
action+, 57
action, 57, 66
alert_cpa_range, 125
alert_range, 124
alert, 122, 123, 128
allow_echo_types, 329
allow_retractions, 139
alt_nav_name, 80, 82
alt_nav_prefix, 80, 82
app_tick, 80
appcast_color_scheme, 55, 65, 158
appcast_font_size, 55, 65, 158
appcast_height, 54, 65, 158
appcast_viewable, 35, 37, 53, 65
appcast_width, 54, 65
apptick, 85, 150
aspect_max, 271
back_shade, 61
beacon, 310, 314
bearing_lines_viewable, 64
behaviors_concise, 74
blackout_interval, 81, 82
blackout_variance, 81, 82
bridge, 218, 221, 229
buoyancy_rate, 142, 150, 151
button_four, 66
button_one, 39, 66, 294
button_three, 66
button_two, 66
center_view, 52, 64
circle_viewable_all, 62
circle_viewable_labels, 62
cmd, 198
collision_range, 338
colormap, 95
comms_pulse_viewable_all, 62
comms_range, 233, 239
comss_range, 233
condition, 108, 113, 188, 190, 342
contact_local_coords, 126, 129
contact_max_age, 126, 129
content_mode, 95
critical_range, 233–235, 239
cross_fill_policy, 80, 82
current_field_active, 143, 201
current_field, 142, 201
datum_color, 62
datum_size, 62
datum_viewable, 62
debug, 239
default_alert_range, 122, 129
default_beacon_color, 310
default_beacon_freq, 310
default_beacon_report_range, 310
default_beacon_shape, 310
default_beacon_width, 310
default_benign_color, 275
default_benign_shape, 275
default_benign_width, 275
default_cpa_range_color, 129
default_cpa_range, 129
default_hazard_color, 275
default_hazard_shape, 275
default_hazard_width, 275
default_hostip_force, 178
default_hostip, 178
default_water_depth, 143, 151
delay_reset, 111, 113
delay_restart, 120

delay_start, 111, 114, 118
delay_time_to_clear, 338
deploy_delay_time, 338
dipplay_virgins, 97
display_all, 95, 97
display_aux_source, 95, 97
display_bhv_posts, 74
display_community, 95
display_empty_strings, 90
display_history_var, 92, 95, 97
display_moos_scope, 74
display_radii, 123, 129
display_source, 95, 101
display_time, 95
display_virgins, 74, 90, 95
drift_vector, 143, 152
drift_x, 143, 152
drift_y, 143, 152
drop_point_coords, 49, 63
drop_point_vertex_size, 63
drop_point_viewable_all, 49, 62
earange, 234, 235, 239
echo_latest_only, 189–191
echo, 186, 189, 190
event, 106, 108, 109, 114, 115
exponential_decay, 329
flip, 187, 190
forward_var, 108, 114
full_screen, 37, 42, 61
grid_config:, 195
grid_config, 194
grid_viewable_all, 63
grid_viewable_labels, 63
grid_viewable_opacity, 63
ground_true, 329
ground_truth, 310, 325
groups, 233, 239
hash_delta, 61
hash_shade, 42, 62
hash_viewable, 42, 62
hazard_file=hazards.txt, 269
hazard_file, 275, 295
history_var, 92, 95
history, 253
hold_messages, 188, 190, 191
keyword, 218
layout:, 248
lclick_ix_start, 59, 66
left_context, 58, 66
log_the_image, 42, 62, 67
marker_edge_width, 63
marker_scale, 63
marker_viewable_all, 63
marker_viewable_labels, 63
marker, 63
max_acceleration, 143
max_deceleration, 143
max_depth_rate_speed, 143, 150, 151
max_depth_rate, 143, 150, 151
max_msg_length, 236, 239
max_time, 293, 295
max_turn_rate, 274, 275
min_acceleration, 147
min_classify_interval, 259, 262, 263, 275
min_msg_interval, 236, 239
min_reset_interval, 263, 266, 275
node_report_output, 78, 82
nodes_font_size, 55, 65, 158
nohelm_threshold, 79, 82
nowatch, 135, 138, 139
nowatcth, 162
oparea_viewable_all, 63
oparea_viewable_labels, 63
oparea, 66
options_summary_interval, 266, 275
pause_var, 107, 108, 114
paused, 74, 107, 114
pd, 286
penalty_false_alarm, 295
penalty_max_time_overage, 293, 295
penalty_max_time_over, 293, 295
penalty_missed_hazard, 295
ping_color, 329
ping_payments, 310, 317
ping_wait, 310, 329
plat_report_input, 82
plat_report_output, 82
platform_length_src, 79
platform_length, 82
platform_type, 82

point_viewable_all, 63
point_viewable_labels, 63
polygon_viewable_all, 63
polygon_viewable_labels, 63
post_mapping, 139, 140
prefix, 143, 189
procs_font_size, 55, 65, 158
publish_immediately, 339
publish_pairs, 339
publish_single, 339
pulse_duration, 338
pulse_range, 338
pulse, 338
qbridge, 227, 229
rand_var, 110, 114
range_pulse_viewable_all, 63
rclick_ix_start, 59, 66
reach_distance, 310, 329
refresh_mode, 55, 65, 89, 95, 158
reply_color, 329
reply_distance = 100, 326
reply_distance, 329
report_vars, 310, 325, 329
reset_max, 107, 114, 118, 120
reset_time, 106, 114, 118, 120
reset_var, 106, 114
right_context, 58, 66
rn_algorithm, 310, 329
rn_uniform_pct, 317
rotate_speed, 143, 149
scope:, 248
scope, 56, 66, 249
script_atomic, 108, 114
script_name, 112, 114
seed_random, 275
seglist_viewable_all, 63
seglist_viewable_labels, 63
sensor_arcs, 329
sensor_config, 275, 279
sensor_pd, 286
show_detections, 275
show_hazards, 275
show_pd, 275
show_pfa, 275
show_swath, 276
shuffle, 106, 114
sim_pause, 143
source, 95, 98, 104
stale_remove_thresh, 51, 64
stale_report_thresh, 51, 64
stale_time, 233, 239
start_depth, 143, 146
start_heading, 143, 146
start_pos, 143, 146
start_speed, 143, 146
start_x, 143, 146
start_y, 143, 146
status_var, 112, 114
stealth, 234, 239
strict_addressing, 243
summary_wait, 137, 139
swath_length, 276
swath_transparency, 276
swath_width, 286
temp_file_dir, 178, 180
term_report_interval, 89, 95, 98
thrust_factor, 143
thrust_map, 143, 154, 156
thrust_reflect, 143, 155, 156
tiff_file_b, 62
tiff_file, 62
tiff_type, 62
tiff_viewable, 62
time_warp, 114, 120
trail_color, 53
trails_color, 64
trails_connect_viewable, 53, 64
trails_length, 53, 64
trails_point_size, 53, 64
trails_viewable, 52, 64
trunc_data, 95, 98
try_shore_host, 218
truncated_output, 74
turn_loss, 143, 147
turn_rate, 143
upon_awake, 107, 114
var, 74, 95, 99, 104
vector_viewable_all, 63
vector_viewable_labels, 64
vehicles_active_color, 64

`vehicles_inactive_color`, 64
`vehicles_name_color`, 51, 65
`vehicles_name_mode`, 65
`vehicles_shape_scale`, 65
`vehicles_viewable`, 65
`verbose`, 114, 239, 243, 310
`view_center`, 62
`view_node_report_pulses`, 47
`view_node_rpt_pulses`, 239
`watch_all`, 135, 139
`watch`, 135, 139, 140