MOOS - Mission Orientated Operating Suite

Paul Michael Newman

Department of Ocean Engineering
Massachusetts Institute of Technology

Abstract

This paper is about simple to use, extensible software for mobile robotic research. It is concerned with a project called MOOS – an acronym for Mission Oriented Operating Suite. MOOS is an umbrella term for a set of libraries and applications designed to facilitate research in the mobile robotic domain. The spectrum of functionality provided ranges over low-level, multi-platform communications, dynamic control, high precision navigation and path planning, concurrent mission task arbitration and execution, mission logging and playback.

The first part of the paper describes the underlying philosophy of MOOS and the resulting perceived benefits. The work then moves on to describe the details of the design and implementations of core system components. There then follows a set of high level descriptions of principal mission-oriented MOOS processes. Collectively these processes constitute a resilient, distributed and coordinated suite of software suitable for in-the-field deployment of sub-sea and land research robots.

Thanks

First let me thank John for his trust and for believing in and paying for the MOOS - I've had a ball. The Sea Grant crew made all the difference, Rob for his advice and unending healthy cynicism, Jim for his tenacity and book of spells , Sam for his skill and amusing use of the "pinger", Joe for great help before, during, and after trials and Justin for Florida and some great backing. I very much appreciate their help, enthusiasm and support. Mike Bosse was a superb client and colleague and always complained about all the right things - cheers. Richard, thanks for your help and being a great mate out here. Watch out for flat inert surfaces - they can hurt a man.

Finally thanks to Sarah. For being great despite too many late nights at MIT and the occasional "lost" weekend, for supplying cookies and ridiculous MOOS paraphernalia. But most of all for having made "Boston" sane and wonderful. Thank you.

Contents

Part I
Introduction

1 Philosophy and Background

MOOS (pronounced "moose") is an umbrella term applying to a set of communicating applications. At its most general the "MOOS" refers to a suite of libraries and executables designed and proven to run a field robot in sub-sea and land domains. Included in its scope are a platform-independent, inter-process communication API, sensor management such as DVL, LBL and SAS sonar, state of the art navigation, vehicle dynamic control, concurrent mission task execution, vehicle safety management, mission logging, mission replay and reprocessing.

The heart of MOOS, its communications API and Library, was written over a period of three days in late September 2001. It is the fourth generation of communications software intended for mobile robot systems written by the author. The design of MOOS and the functionality provided by its communications API was motivated by experience and observations gained in using and developing several software suites for mobile robots. In particular it was noted that:

- Dependence on a particular operating system is bad news. The development tools available in one choice may far out-perform those available in another. Conversely the opposite may be true in terms of real-time performance and networking. No one choice satisfies all criteria.

- Allowing source code dependencies to occur between different applications can lead to project management problems and slowed development. As the complexity of the project grows so does the number of inter-dependencies and time spent coordinating rebuilds etc.

- Researchers often require a sense of ownership over an element of a robotic system. It is desirable to be able to have an individual write code in his or her own style and in isolation and yet still have the resulting application interact with any number of other applications/services running on the vehicle.

With these observations in mind the following design goals were set:

- Platform independence - Code should run on Linux, NT, and Win2000.

- A collection of communicating processes should run the vehicle - each process encapsulating a specialization or core functionality.

- The communications provided to these processes should be utterly robust and tolerant of repeated stop/start cycling of any process.

- Processes should have no shared header file dependencies.

2 Project Management in the Research Domain

It is often the case that executables that need to interact with each other have a clear 'owner', i.e have been developed on the whole by a single researcher/engineer. Inevitably then, this work carries the unique stylistic and architectural finger-prints of the creator. This development model can be efficient until several such projects are merged into a monolithic application. It is at this point that code strain starts to show. Much effort can be expended in fusing the projects with no overall change in functionality. MOOS has been designed to allow projects to be melded without blurring or dissolving process boundaries - the number of processes remains the same. Instead each process joins a "community" of MOOS-enabled applications. Engineers can keep coding their own application independently of all other source code. This has ramifications when considering versioning and source code control. The projects can live completely independent lives - there is no compiled code dependency between them. A code change in one domain cannot cause compilation failures in another.

Part II
Foundations

3 Overview

It is useful to provide a brief description of a typical MOOS driven system to provide a compelling motivation for the rest of this document.

Consider the case of an single AUV (autonomous underwater vehicle) operating within an LBL (long base-line) acoustic array [1]. MOOS provides all the software components and communications required manage sensors, navigate, control actuators, plan trajectories, monitor safety , log performance and both manual and automatic control. Figure 3 shows a typical consortium of processes used on a typical AUV research mission. In fact this setup was used during the "GOATS 2002" experiment in Italy 2002. The AUV was carrying a synthetic aperture sonar managed by a separate PC in a payload section. MOOS enabled seem-less integration of this payload with the processes running on the main vehicle computer while providing real-time navigation and control of the vehicle using a multitude of sensors. The details of these processes and more importantly the underlying infrastructure will be discussed in future sections of this document. At this juncture the most important message to take home is that MOOS is a designed to provide:

1. A standardized, multi-platform way for processes to share information
2. A set of key processes that fulfill ubiquitous roles in mobile robotics

4 Topology

MOOS has a star-like topology. Each application within a MOOS community (a `MOOSApp`) has a connection to a single "MOOS Database" (called `MOOSDB`) that lies at the heart of the software suite. All communication happens via this central "server" application. The network has the following properties:

- No Peer to Peer communication.
- All communication between the client and server is instigated by the client. (ie the `MOOSDB`**never** makes a unsolicited attempt to contact a `MOOSApp` .)
- Each client has a unique name.
- A given client need have no knowledge of what other clients exist.

[1] MOOS can and has been equally well applied to land robots however the AUV example is pretty interesting

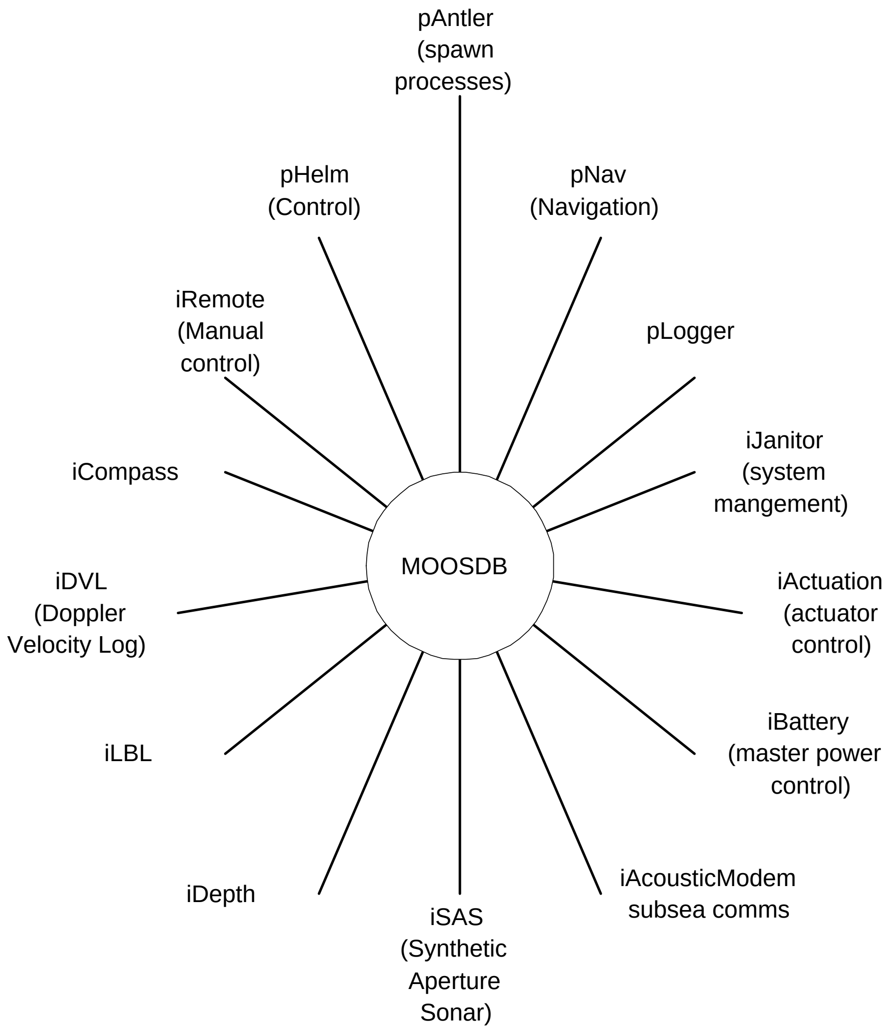

Figure 1: A complete MOOS configuration for AUV deployment. Taken from the "GOATS 2002" experiment, Italy 2002

- A client has no way of transmitting data to a given client - it can only be sent to the `MOOSDB`.
- The network can be distributed over any number of machines running any combination of supported operating systems.

This centralized topology is obviously vulnerable to "bottle-necking" at the server regardless of how well written the server is. However the advantages of such a design are greater than its disadvantages. Firstly the network remains simple regardless of the number of participating clients. The server has complete knowledge of all active connections and can take responsibility for the allocation of communication resources. The clients operate independently with inter-connections. This prevents rogue clients (badly written or hung) from directly interfering with other clients.

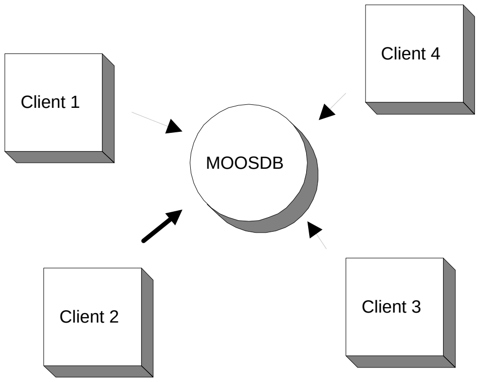

Figure 2: MOOS binds applications into a network with a star-shaped topology. Each client has a single communications channel to a server (MOOSDB).

Variable	**Meaning**
Name	The name of the data
String Val	Data in string format
Double Val	Numeric double float data
Source	Name of client that sent this data to the MOOSDB
Time	Time at which the data was written
Data Type	Type of data (STRING or DOUBLE)
Message Type	Type of Message (usually NOTIFICATION)

Table 1: Contents of MOOS Message

5 Message Content

The communications API in MOOS allows data to be transmitted between MOOSDBand a client. The meaning of that data is dependent on the role of the client. However the form of that data is constrained by MOOS. Somewhat unusually MOOS only allows for data to be sent in string or double form. Data is packed into messages (CMOOSMsg class) which contains other salient information shown in Table 1. The fact that data is commonly sent in string format is often seen as a strange and inefficient aspect of MOOS. For example the string `Type=EST,Name=AUV,Pos=[3x1]{3.4,6.3,-0.23}` might describe the position estimate of a vehicle called “AUV” as a 3x1 column vector[2]. It is true that using custom binary data formats does decrease the number of bytes sent. However binary data is unreadable to humans and requires structure declarations to decode it and header file dependencies are to be avoided where possible. The communications efficiency argument is not as compelling as one may initially think. The CPU cost invoked in sending a TCP/IP packet is largely independent of size up to about one thousand bytes. So it is as costly to send two bytes as it is one thousand. In this light there is basically no penalty in using strings. There is however a additional cost incurred in parsing string data which is far in excess of that incurred when simply casting binary

[2]Typically string data in MOOS is a concatenation of comma separated ”name = value” pairs.

data. Irrespective of this, experience has shown that the benefits of using strings far outweighs the difficulties. In particular:

- Strings are human readable - debugging is trivial especially using a tool like MOOSScope. (see Section 20)
- All data becomes the same type
- Logging files are human readable (they can be compressed for storage).
- Replaying a log file is simply a case of reading strings from a file and "throwing" them back at the `MOOSDB`in time order.
- The contents and internal order of strings transmitted by an application can be changed without the need to recompile consumers (subscribers to that data) - users simply would not understand new data fields but they would not crash.

Of course, scalar data need not be transmitted in string format - for example the depth of a sub-sea vehicle. In this case the data would be sent while setting the data type to `MOOS_DOUBLE` and writing the numeric value in the double data field of the message.

6 Threading and Process Models

The choice of processes over threads was made on two counts, firstly that of stability - a rogue process cannot corrupt the program/data space of another process (in a sane OS that is). Secondly on the basis of swift and pain-free development by several programmers with diverse backgrounds. Building a single monolithic executable by several people requires, at a minimum, adherence to programming guidelines and styles that may not be native to all those included - especially in an academic environment. The use of small-footprint, independent processes implies developers can use whatever means they see fit to accomplish the job. Linking with the communications library integrates them seamlessly with all other processes but denies a process the means of interfering with others.

7 Communications Mechanics

Each client has a connection to the DB. This connection is made on the client side by instantiating a class provided in the core `MOOSLIB` library called `CMOOSCommClient`. This class manages a private thread that coordinates the communication with the `MOOSDB`. The `CMOOSCommClient` object completely hides the intricacies and timings of the communications from the rest of the application and provides a small, well defined set of methods to handle data transfer. Using the `CMOOSCommClient` each application can:

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

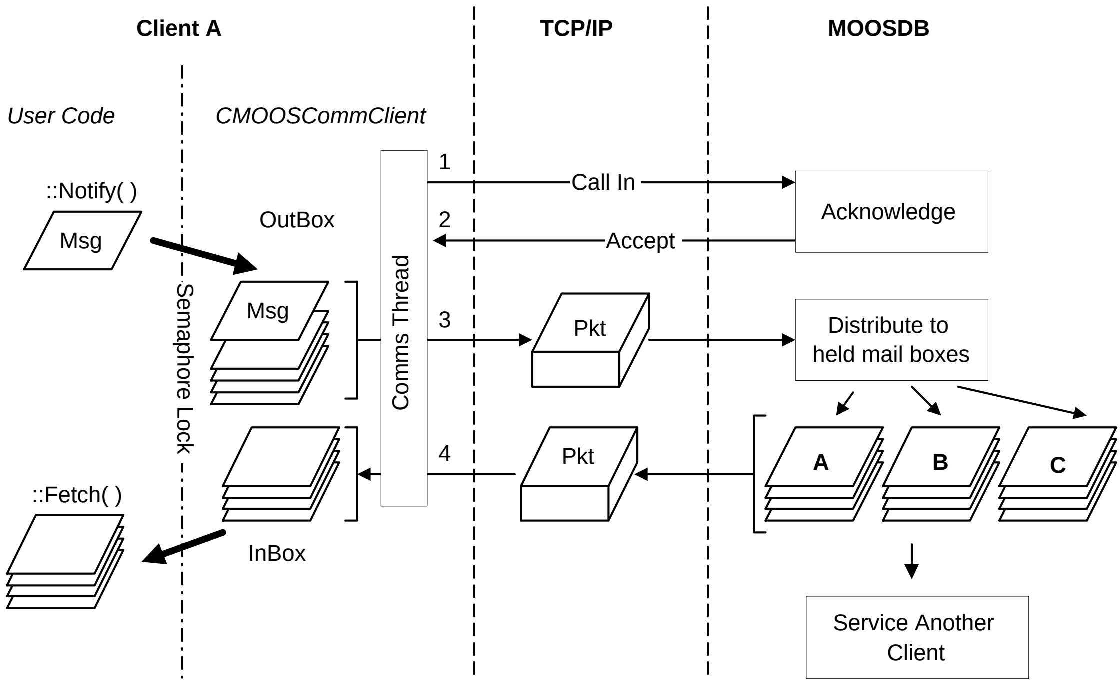

Figure 3: The mechanics of the client server interaction in MOOS. The user code calls the `Notify` method to transmit data. This method simply adds a message to the "outbox". Some time later (1) the communications thread calls into the database. When the database is not busy it accepts the client's call (2). The client then packs the entire outbox into a single large transmission which is sent to and read by the server (3). The server unpacks the packet into its constituent messages and place copies (according to subscriptions and timing) in the mailboxes of other connected clients. The server then compresses the mailbox of the current client into a packet and sends it back to the client (4). At this point the transaction is then complete and the server terminates the conversation and looks to begin the same process with a different client. Upon receiving the reply packet the client communications thread unpacks it and places the resulting messages in the "inbox" of the client. The user code can retrieve this list of messages at *any time* by calling the `Fetch` method.

7.1 Publishing Data - notification

Assume as a result of some computation or input, a process `A` has a result that is likely to be of use other undisclosed (remember MOOS applications do not know about each other) processes `B` and `C` - for example a position estimate. The simplest way in which `A` can transmit its new data is to simply call the `Notify` method on its local `CMOOSCommClient` object specifying the data, its name and the time at which it is valid. Behind the scenes this method then creates a suitable `CMOOSMsg` filling in the relevant fields. The action of publishing data in this way should be viewed as a *notification* on a named data variable. Crucially this does not imply a change in the *value* - the outcome of the computations resulting in the need to publish data may remain not change . For example two sequential estimates of location of a vehicle may remain numerically the same but the *time* at which they are valid changes - this constitutes a data notification.

As far as the application-specific code in A is concerned the invocation of the Notify method results in the data being sent to the MOOSDB. What is really happening however is that the CMOOSCommClient object is placing the data in an "outbox" of CMOOSMsgs that need to be sent to the MOOSDBat the next available opportunity. The next obvious question is "when does the data reach the MOOSDB?". The CMOOSCommClient object thread has a "Comms Tick" - essentially a timer that contacts the MOOSDBat a configurable rate (typically 10Hz but up to 50Hz). All the messages in the outbox are packed into a single packet or "super message" called a CMOOSPkt. Eventually the MOOSDBwill accept the incoming call from the client A and receive the packet. At this point the data flow is reversed and the MOOSDBreplies with another CMOOSPkt containing notifications (issued by other processes like B and C) that are relevant to A. How a process declares what constitutes a relevant notification is discussed in section 7.2. If at the time the thread calls into the MOOSDBthe outbox is empty (i.e there is nothing to notify) a NULL message is created and sent. Similarly the MOOSDBmy reply with a CMOOSPkt containing only a NULL message if nothing of interest to the client has happened since its last call in. This policy preserves the strict symmetry of "one packet sent, one packet received" for all occasions.

7.2 Registration

Assume that a list of names of data published has been provided by the author of a particular MOOS application. For example, a application that interfaces to a GPS sensor may publish data called GPS_X and GPS_Y. A different application may register its interest in this data by *subscribing* or *registering* for it. An application can register for notifications using a single method Register specifying both the name of the data and the maximum rate at which the client would like to be informed that the data has been changed. The latter parameter is specified in terms of the minimum possible time between notifications for a named variable. For example setting it to zero would result in the client receiving in the CMOOSPkt (that is the MOOSDB's reply) a collection of messages describing each and every change notification issued on that variable.

7.3 Collecting Notifications

At any time the owner of a CMOOSCommClient object can enquire whether it has received any new notifications from the MOOSDBby invoking the Fetch method. The function fills in a list of notification CMOOSMsgs describing what has changed, what client made the change, when it was done and what the data type is (see table 1 in section 5). Note that a single call to Fetch may result in being presented with several notifications corresponding to the same named data. This implies that several changes were made to the data since the last client-server conversation. However, the time difference between these similar messages will never be less than that specified in the Register function described in section 7.2.

8 The MOOSDB

The `MOOSDB`is the heart of the system. It serves as the hub through which all communication occurs. It is tempting to think of the `MOOSDB`as simply a blackboard - an entity which stores the current state (represented by values of named variables) of the system. Typically a blackboard allows clients shared access to a centralized repository of information. Although the `MOOSDB`does maintain the most recently set value of all variables and in that way is similar to a blackboard, the way in which data is retrieved is very different. The `MOOSDB`records, on behalf of its connected clients, the *history* of changes to data. Assume a client `A` has subscribed to variables called `p` and `q` with a minimum notification period of τ_p and τ_q seconds respectively. The time is now t and the last call-in was at time t',$\Delta = t - t'$ seconds ago. When `A` calls in, it is not simply presented with the most recent values of `p` and `q` but rather all the *changes* that have occurred to them between t' and t'. Imagine that some other process `B` is publishing changes (value and/or time-stamp) to p and q every τ_B seconds. The call in will result in n_p and n_q notification messages being returned to the client where

$$n_p = \begin{cases} floor(\frac{\Delta}{\tau_p}) & \tau_p \geq \tau_B, \\ floor(\frac{\Delta}{\tau_B}) & \text{otherwise.} \end{cases}$$

$$n_q = \begin{cases} floor(\frac{\Delta}{\tau_q}) & \tau_q \geq \tau_B, \\ floor(\frac{\Delta}{\tau_B}) & \text{otherwise.} \end{cases}$$

If no clients have issued relevant notifications since t' then there will be no notification messages stored at the `MOOSDB`for collection by the client. The only exception to this rule is the first time a client calls in after registering for a notification on a variable. In this case the value of the variable is returned in a notification message with a time-stamp specifying when the data was last set.

Part III
Using The MOOS – Libraries and Classes

The combination of the MOOSDBand the two libraries MOOSLib and MOOSGenLib are all that is needed to build a MOOS community. The role and operation of the MOOSDBhas already been discussed in section 8. The contents and utilities provided by the libraries are now described.

9 MOOSLib

The primary role of MOOSLib is to contain all the communications components used both by the MOOSDBitself and CMOOSCommClient objects owned and used by client applications.

9.1 CMOOSApp

Perhaps the most important class exposed from the library is CMOOSApp. This class should be used as a base class for all MOOS applications. It provides along with other things:

- Management and configurations of a CMOOSCommClient object
- Tools for reading configuration parameters (using a file reading tool exported from MOOSGenLib).
- Timing control of the main thread of the application and an additional communications thread.
- Virtual functions that can be overloaded to perform specific actions when:
  1. New mail (notifications) arrives
  2. The default work of the application should occur
  3. The client connects to the MOOSDB
  4. The client disconnects from the server
  5. The application is about to start

Appendix A gives example code of a skeleton MOOS application using the functionality provided by deriving from CMOOSApp. Indeed this code could be used as the template from which to start building all new MOOS applications. The timing of the application is determined by two timer frequencies AppTick and CommsTick which are set in the process configuration file (see section 10.1). The CMOOSApp base class does its best to call the virtual Iterate method at the frequency specified by AppTick. It is within this method that the bulk of the work of the application should be undertaken - for example processing data or executing control loops. Just before `Iterate` is called the `CMOOSApp` base class calls `Fetch` (see section 7.3 on its `CMOOSCommClient`. If mail is present then the virtual function `OnNewMail` is called. Here an overloaded version is passed a list of `CMOOSMsg`s which can contains the relevant notifications from the `MOOSDB`and which can be acted upon as the application sees fit. Only messages corresponding to variables that have been registered for will be present. Typically an application registers its interest in named data by calling `CMOOSCommClient::Register()` during the virtual callback `OnConnectToServer()`. This guarantees that the server has been found, successfully connected to and subsequently any registrations shall be successful. It should be noted that attempting to register outside of this function (for example in the overloaded `OnStartUp` method) may fail because the communications thread with `CMOOSCommClient` has not yet established a connection with the `MOOSDB`.

9.2 `CMOOSInstrument`

The class `CMOOSInstrument` is another important base class. It is intended to simplify the writing of applications interacting with hardware via a single serial port. The class extends `CMOOSApp` with utilities to manage and set up a platform-independent serial port[3]. The serial port can be configured to be asynchronous and receive unsolicited streaming data or to be synchronous and perform blocking read and writes. The choice depends on the device the application is intended to communicate with.

10 MOOSGenLib

The library `MOOSGenLib` is a tool chest. It contains utilities and classes used throughout MOOS. In particular it provides:

- Platform-independent serial ports (asynchronous and synchronous).
- Thread safe configuration reading tool – `CMOOSMissionFileReader`.
- String manipulation/parsing tools.
- Geodesy tools.
- debug statement tools - MOOS equivalent of `printf`.

Of these assets the configuration file reader in particular warrants more discussion.

10.1 Configuration Files *.moos

Every MOOS process can read configuration parameters from a "Mission file" which by convention has a ".moos" extension. For example the default mission file mentioned in

[3]The serial port classes live in `MOOSGenLib` - see section 10.

Figure 4: A typical configuration block for a MOOS application. A process called "iDepth" will search a mission file until a block like this is found. It will then parse our configuration parameters.

```
/////////////////////////////
// depth sensor configuration
ProcessConfig = iDepth
{
    AppTick = 8
    CommsTick = 4
    Port        = com1
    BaudRate    = 9600
    Streaming   = false
    Type        = ParaSci
    Resolution  = 0.1
}
```

the example code given in Appendix A is *Mission.moos*. Traditionally MOOS processes share the same mission file to the maximum extent possible. For example it is usual for there to be one common mission file for all MOOS processes running on a given machine. Every MOOS process has information contained in a configuration block within a *.moos file. The block begins with the statement

`ProcessConfig` = `ProcessName`

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

Each statement is written as

`ParameterName = Value`

where `Value` can be any string or numeric value. Typically the format of the value string is a comma separated list of strings or numbers. However some applications may require more complicated syntax using a combination of delimiters other than simple commas. Of course, the details of these settings is part of the application documentation. Examples can be found on the MOOS website (*http://oceanai.mit.edu/pnewman*). All applications deriving from `CMOOSApp` and `CMOOSInstrument` inherit several important configuration options. For example, `CommTick` and `AppTick` (see section 9.1. The documentation on the MOOS webpage gives details of these common parameters.

Figure 10.1 gives an example of a typical configuration block, in this case for a depth sensor. The parameters `Type` and `Resolution` are specific to the class defining the methods of a "DepthSensor". All the other parameters are handled by its base classes (`CMOOSInstrument` and `CMOOSApp`).

10.2 Utility Functions

MOOSGenLib contains a host of utility functions that are described below. These functions are ubiquitous within MOOS and should not be substituted with local version producing the same functionality. The reason is the innards if these functions may be built to accommodate subtleties in MOOS. A good example is that of MOOSTime which returns the current process time *corrected* by a constant term setup by the MOOS communications when the server is contacted.

MOOSFormat The MOOS version of sprintf. It returns a formatted std::string object.

MOOSTrace The MOOS equivalent of printf printing a formatted string to the console. When run under win32 the function also writes to a debugger window if applicable. Note that MOOSTrace does not use cout because on some Linux releases this is not thread safe.

MOOSChomp A massively useful, simple string parsing function. MOOSChomp is passed a string, S, to act on and a delimiter string, D, as parameters. It returns the substring occurring before D in S and removes everything up to the end of the D from the S. For example if S = "X = 4, Y = 9" and D = "," then calling MOOSChomp(S,D) would return "X = 4" and S would now be "Y=9".

MOOSTime Returns the current time in decimal seconds (a double) for the current process. All connected processes will show the same time *even* if their respective machine clocks differ. This is achieved by deducing a client correction during initial handshaking with the MOOSDB.

MOOSGetTimeStampString returns a time/date string formatted in the MOOS convention - useful for naming temporary local files for development purposes etc.

MOOS_ANGLE_WRAP Wraps all angles (in radians) to be with $\pm\pi$ - forgetting to wrap angles can cause woe.

MOOSPause Pauses the current thread (not process) for a specified number of milliseconds.

10.3 Serial Ports

The CMOOSInstrument class extends the CMOOSApp class to offer non-blocking serial port functionality. The serial port operations can be one of two modes - synchronous and asynchronous.

Synchronous This mode is designed to be applied to instruments that are synchronous - that is they only talk when asked to. When in this mode the serial port owner is responsible for sending commands and listening (with a timeout) for replies as required. It is the simplest mode and commonly employed.

Asynchronous / Streaming This mode is designed for use with instruments that issue a stream of unsolicited transmissions. In this case the serial port class manages a listen thread that parses incoming bytes into a series of telegrams, which are stored along with a time stamp in a list. The characters/bytes that delimit an end of telegram in the input stream can be changed on the fly. At any time the serial port user can query the store of telegrams to retrieve a message or perhaps purge the list of its contents.

Part IV
MOOS Conventions

11 Coordinate Conventions

MOOS processes communicate using a defined coordinate system illustrated in figure 11. The salient points to note about this system are:

- The global (earth) frame is a conventional East-North-Up frame.
- The vehicle body frame has been designed to align with the global frame when the vehicle has zero yaw.
- At zero heading the vehicle points north.
- Zero heading is equivalent to zero yaw.
- Yaw = -heading*π/180 (heading is in degrees).
- Depth is in the opposite sense to Z

The fact that a coordinate system is defined that must be used between processes *does not mean* this system is inflicted on the internals of an application. Within the boundaries of a process the developer is free to use whatever system he or she feels comfortable with. All that is needed is a little patch work to transform input and output data between coordinate frames. However life can be made easier by using one coordinate frame throughout.

12 Geodesy

The geodetics with MOOS are refered to as `MOOSGrid`. The geodesy tools within MOOS assume operation in a local area - say a square with sides less than 20 km long. With this assumption it is reasonable to superimpose a local cartesian space over the work area and form a mapping from Lat/Long coordinates to local `MOOSGrid`. The class `CMOOSGeodesy` in `MOOSGenLib` performs all the mapping required. It is primed with the origin of the `MOOSGrid` (in decimal lat/long). It can then convert lat/long measurements to cartesian `MOOSGrid` coordinates using a geodetic radius that is a function of the origin location.

13 Units

MOOS was designed and built by a European and so uses easy SI units throughout. All distances are in meters and times in seconds. Angles are always in radians and wrapped

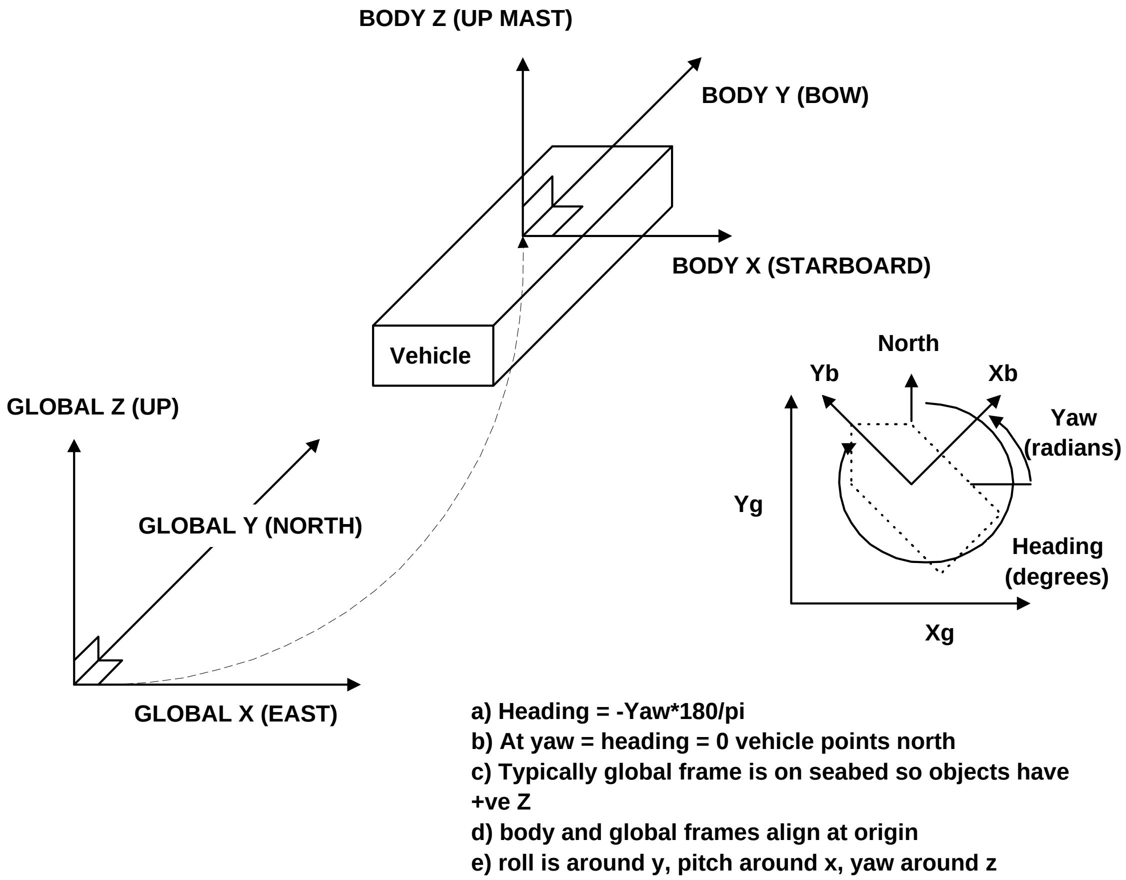

Figure 5: The MOOS coordinate system. Note that $\dot{yaw} = -heading$ and that the body and earth frames align at the origin with the vehicle pointing north at zero yaw. Also the body 'y' axis is towards the front of the vehicle. (You may be used to having the body x axis point towards the nose of the vehicle). Note also that Z points up.

to lie with $\pm\pi$. The only exception to this is "heading" variables, which are in degrees. However heading variables are never used internally - they are for human display alone. Instead their "Yaw" counterparts are used, which are always in radians.

14 Naming Conventions

14.1 Process Naming

MOOS developers are encouraged to adopt the naming convention for applications given in table 14.1. The basic idea is that the prefix to the process gives some indication of its function. Processes beginning with "i" are typically derived off `CMOOSInstrument` and perform some kind of I/O via serial port (often to a hardware sensor) or non-MOOS communications. Processes beginning with "p" are pure and perform no I/O other than via the MOOSDB. Finally "u" processes are utilities - performing tasks not critical to

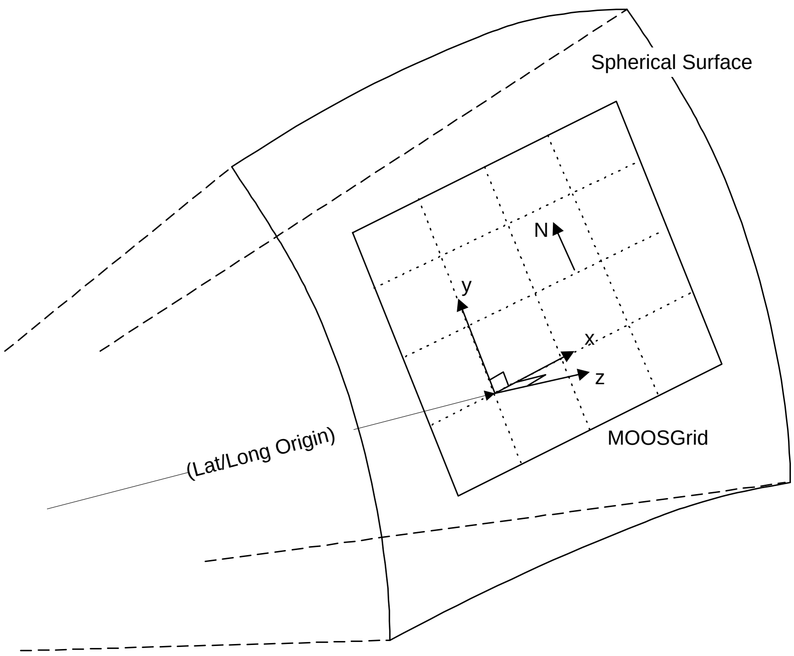

Figure 6: Defining a local cartesian `MOOSGrid` around a lat/long origin

vehicle operation but which are perhaps useful at other times - for example `uGeodesy` converts from “Lat/Long” to local-grid (`MOOSGrid`) coordinates.

Table 2: The naming convention for MOOS applications

Name	**Description**	**Example**
i[*Name*]	Interface applications. Interacts (has I/O) with an external device, for example via a keyboard or serial port	`iGPS, iCompass, iRemote`
p[*Name*]	Pure applications. Only interacts with other MOOS applications	`pHelm, pLogger, pNav`
u[*Name*]	Utility applications. Not used at run time but a useful at other times	`uPlayBack, uGeodesy`

14.2 Variable Naming

Data can take any string name. A developer could use any combination of characters to name the data published by his or her application. In practice though sticking to a convention makes some things easier. A good example of this is the `pNav` process that expects some kinds of sensor data to be named in a standard format. Sensor data can be classified into at least the following categories:

If a sensor, managed by a process called `iSensor`, measures one of these quantities then the name under which the data should be published has the format `SENSOR`_*CATEGORY*. This is best highlighted with a few examples:

Quantity	Description	Example
X,Y,Z	sensor measures vehicle position	GPS
DEPTH	sensor gives depth	Pressure sensor
YAW	sensor measures rotation around Z axis	Gyro
BODY_VEL	sensor measures velocity in body frame	DVL, Odometry

Table 3: Sensor categories

- iGPS measures X and Y position. It publishes GPS_X and GPS_Y.
- iDepth measures depth. It publishes DEPTH_DEPTH.
- iLBL measures range and depth. It publishes LBL_DEPTH and LBL_TOF (time of flight).

This simple convention makes visual comprehension of the system (when using a tool like MOOSScope 20) simple. Additionally it avoids naming conflicts as the MOOSDBrequires unique process names and therefore basing data names on process names will also result in unique data names.

14.3 Actuator Naming

At the time of writing MOOS applications are aware of three actuator types: THRUST, ELEVATOR and RUDDER. Clearly this illustrates the early history of MOOS as software for autonomous underwater vehicles (AUVs). On land the ELEVATOR actuator has no meaning. The THRUST and RUDDER axes however map well to throttle and steer angle.

Note that perhaps counter intuitively, but consistent with the MOOS coordinate frame, a positive elevator angle will cause a vehicle to make its pitch more negative. Similarly positive rudder will make yaw decrease (but heading increase).

Part V
Key MOOS Processes

This section describes some of the MOOS applications that have been written using the MOOS API and libraries for use on mobile robots. This section is not intended to provide definitive documentation on these processes but rather to provide a good summary of a field-tested installation of a MOOS community.

15 The Helm - `pHelm`

Along with `pNav` the Helm process `Helm` is one of the most important high-level processes that are typically run on a given mission. The Helm's job is to take `NAV_*` from the navigator and, given a set of mission goals, decide on the most suitable actuation commands. The multiple mission goals take the form of prioritized tasks within the Helm. For example a "snap-shot" of the helm might reveal five active tasks: follow a track line, stay at constant depth, limit depth, limit altitude and limit mission length. The first two of these are conventional trajectory control task that could be expected to be found in any mobile robot lexicon. The last three tasks are safety tasks that take some action if a limiting condition specified within them is violated.

The Helm is designed to allow "at sea" reloading of missions. If the mission file (specifying what tasks to run) is edited while the Helm is runnning, the Helm can be commanded via a MOOS message (`RESTART_HELM`) to clean down and rebuild itself. This makes for very rapid turnaround of missions in a research-oriented field trip.

The Helm has two modes: online and off-line. When off-line no tasks are run and the Helm make no attempt to control actuators. In his mode the vehicle can be controlled via `iRemote` (see 17. When on-line the vehicle's motor are under the control of the Helm - autopilot is engaged. As would be expected `iRemote` is used to send the signal that relinquishes the manual control of the vehicle and puts the Helm online. Manual control can be regained at any time by pressing "space" on the iRemote console.

The details of tasks that can be run by the Helm and topics relating to their management are now discussed.

15.1 Tasks

As might be expected, the Helm and its constituent tasks rely heavily on the communications provided by the lower level MOOS API. The tasks use the communications apparatus to coordinate themselves. The parent Helm application is derived from `CMOOSApp` and as such has access to the MOOS communications system. The Helm application handles all communications on behalf of its owned tasks. All received `CMOOSMsg`s are offered to all active tasks. Each task is queried for any newly required subscriptions that are subscribed for by the Helm. In a similar fashion each task is queried for any messages that it requires to be emitted. This mechanism gives each task the illusion

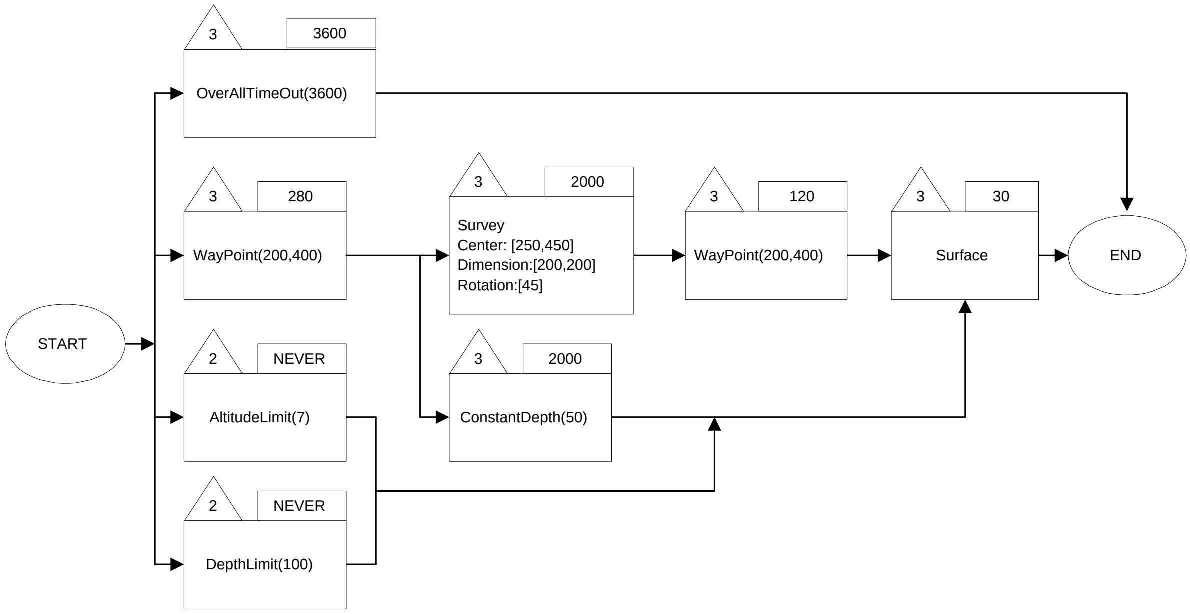

Figure 7: A typical mission plan. Traverse to (200,400), dive to 50m and perform a survey. Impose safety limits of 100m depth and 7m altitude. When the Survey is complete (or timed out) return to the dive co-ordinate and surface. If something terrible has happened and mission is still running after one hour - abort immediately. The task priorities are written in triangles and timeouts in boxes above the task descriptions.

that it has its own `CMOOSCommClient` object even though it is in fact sharing it with other active tasks.

All tasks are derived from a common base class that provides all descendants with a shared heritage. In particular the ability to specify in the mission file some key named properties which are described in Table 15.1. Perhaps the most important thing to understand is that tasks use MOOS messages to synchronize themselves. As one task finishes it emits one or more messages which may be "just the thing" that other tasks were watching for to signal that they should go active. Note that this has two important consequences:

- One task can activate any number of other tasks.
- A task can be activated by activity *outside* of the helm. For example a command received by an acoustic modem could start a task. This open architecture is a powerful concept within MOOS.

15.1.1 Task Completion

A given task completes when any of the following conditions are met:

- It has completed successfully - fulfilled its goal criteria. For example it has steered the vehicle close (enough) to a way point or has driven the vehicle to a specified depth.

Table 4: Base Task Properties

Property	**Use**
Name	The name of the task - for example "Leg7" or "InitialDelay"
StartFlags	A list of messages names that if received, will spur the tasks into action (turn it on).
FinishFlags	A list of messages that are emitted when the task completes or when it starves because it is not receiving notifications on one or more of its subscribed variables.
EventFlags	A list of messages that are emitted when some event happens but the task does not complete. A typical example of this would be a depth limit task emitting event messages when the vehicle exceeds a specified limit. The last thing you want to happen is the task quitting just when this has happened. Instead the event flags are sent and the task keeps running.
TimeOut	The maximum time the task should run for. If the task does not complete naturally before this timeout the FinishFlags are sent and the task retires itself automatically. The value of "NEVER" can be specified to indicated that a task should never timeout — useful for safety tasks.
InitialState	A task can be on initially in which case it does not listen for start-flags. Or it can be off in which case it must receive a start-flag before it goes active.
Priority	Each task is assigned a priority from 1 to ∞ (0 is reserved for the special EndMission task). The lower the value the more important the task is.

- It has timed out before achieving its goal.
- It has starved. This occurs when it is does not receive the data it needs at regular enough intervals. This is a crucial safety feature. Say for example the navigation has failed - NAV_* will not be being emitted by the navigator and a motion control task requiring navigation information to function should not be allowed to continue to run with blithe disregard. Starvation of tasks is a sure sign that something is wrong with the system configuration or navigation.

15.1.2 Task Arbitration

The Helm employs the simplest of strategies in deciding which task wins when two concurrent active tasks are both trying to control the same actuator — the one with highest priority (lowest numerical value) wins. To illustrate this point envision the case where a track-line task is running at priority level three. Then a way-point task with priority level two goes active - perhaps after reception of data via an acoustic modem. The way-point task will take control of the rudder actuator until it completes. At this point the original track-line task will resume. Throughout the episode the track-line task is unaware that it does not have control of the vehicle. In the case that one or more equal priority tasks controlling the same actuator(s) are active the Helm simply chooses to give the first task processed (in its list of active tasks) control.

15.2 Task Configuration and *.hoof Files

Tasks are configured in a similar way to processes — within a named, brace delimited block of text. Any number of tasks can be specified in this fashion. A task configuration block always begins with `Task=`*TaskType*, where *TaskType* is one of the named types in Table 15.2.1 followed by the opening brace of the block. Figure 15.2 shows typical task configuration blocks for two tasks. In this case two task are being configured – one to

```
Task = ConstantHeading
{
    Priority = 3
    Name = South
    Heading = 180
    TimeOut = 300
    InitialState = OFF
    StartFlag = MissionStart
    FinishFlag = GoNorth
    LOGPID = true
}

Task = ConstantHeading
{
    Priority = 3
    Name = North
    Heading = 0
    TimeOut = 120
    InitialState = OFF
    StartFlag = GoNorth
    FinishFlag = EndMission
    LOGPID = true
}
```

drive south for 300 seconds and when this task completes another will drive north for 120 seconds — note the pairing of Start and Finish flags. Each task derived from the base task class is likely to add its own specialization parameters (like `Heading` in the case of `ConstantHeading` Tasks). The semantics and specifications of these additions are found in the task documentation (see the MOOS web-pages http://oceanai.mit.edu/pnewman).

15.2.1 Task Specification Redirection

If the Helm configuration block contains the line "`TaskFile=` *filename.hoof*" then at startup the helm builds all the tasks it finds in the specified `*.hoof` file. This indirection is useful for building a library of missions that can be loaded into the Helm at any time simply by changing a single line in the Mission file.

Table 5: A summary of common task functionality

Task	**Use**
`TimeOut`	Issues `FinishFlags` after timeout. This is useful for creating a pause between tasks - for example a delay at mission start.
`GoToWayPoint`	Go to a specified XY location. A tolerance can be specified to stop un-ending hunting. Requires X,Y and Yaw data.
`ConstantDepth`	Level Flight. Requires only depth data and pitch data.
`ConstantHeading`	Drives the vehicle at a constant heading. Requires only yaw data.
`XYPattern`	Repeats a pattern of way-points. Requires X,Y and Yaw data. The number of repetitions required can be specified.
`ZPattern`	Performs a series of depth set points. Requires depth and pitch data. For example combining this with a orbit task will execute a helical pattern.
`GoToDepth`	Spiral to a given depth and exit. Requires Pitch and Depth data.
`LimitAltitude`	Fire event flags if too close to sea bed. A Safety task requiring Altitude data. This has saved the vehicle several times.
`LimitDepth`	Fires event flag if the vehicle is too deep. A Safety task that is usually run with timeout=NEVER
`EndMission`	Abort Mission (highest priority). Set to be the highest priority task (0). Locks out all other tasks. Usually commands all actuation to zero.
`DiveTask`	Dive from surface (e.g reverse dive)
`Orbit`	Orbit a given location in XY plane. The direction, radius and number of orbits can be specified
`Survey`	Perform a survey (mow lawn) centered on specified position with given rotation and extent.
`TrackLine`	Execute a linear path between two points.
`OverAllTimeOut`	Limit Total length of mission. Always use this task - it is compulsory. Its FinishFlag should be the StartFlag of an EndMission task.
`LimitBox`	Fire event flags if 3D working volume of mission is exceeded. A Safety task useful for trapping navigation failure.

Table 15.2.1 gives a brief summary of the most commonly used tasks found in the `MOOSTaskLib` library. This list is not exhaustive but serves to illustrate the kind of functionality provided. The configuration parameters for each task can be found on the MOOS documentation web-pages.

15.3 Third-party Task Execution

An interesting component of the Helm is its ability to dynamically launch a task on behalf of a another client – "a third-party request". Obviously some security needs to be in place to stop wayward software requesting tasks to be performed that endanger other aspects of the mission or even the vehicle itself. The issue here is that third-party requests are simply MOOS messages with a certain string format issued from the innards of some unknown application. The third-party request security mechanism requires that the human creating the mission file actively grants permissions (by placing certain text in the Helm's configuration block) for the Helm to launch a specified task on behalf of a specified (named) client. Any third-party requests not mentioned in the configuration block will be denied.

15.3.1 Common Usage

One may wonder why given the event-driven nature of the Helm such a scheme is needed. The architecture was implemented by extending the model of Navigator and Helm pair to include a 'Guest Scientist' — a client application which, on the fly, makes requests to the Captain (Helm) to alter course to accommodate the needs of some scientific mission. However,

> "The Captain reserves the right to deny the request on the grounds that it may jeopardize the safety of the vessel or contradict more important mission orders."

This clause summarizes the interacting priorities of existing tasks and the extent of dynamic task generation the Helm is allowed to grant to third-parties.

15.3.2 Granting Permissions

The format of the permission specification in the Helm configuration block is as follows.
`Allow=`*jobname*@*clientname*:*tasktype* | `SessionTimeOut=`*Timeout*
The fields have the following meaning:

Jobname Describes the kind of work that the application requesting the launch of a task wants to do. It can be any name, for example, it might be "Explore" or "Detect Mine" – something that is meaningful to humans.

Clientname is the name of the application making the request, for example, `iAcousticModem`. It is the name that is used by the client when communicating with the DB.

Tasktype is the string name of a task supported by the Helm application. For example `XYPattern`.

SessionTime is the time the client has after completion of a requested third-party task during which it can request another task and be guaranteed that it will be accepted (provided it has the relevant permissions). This is a subtle point: imagine a client has requested the vehicle traverse to a distant way-point; having got there we do not want a competing client to request some other destination before the first task has had a chance to perform whatever it wanted to do at the "distant" way-point. Typically this value is set to a few seconds.

15.3.3 Request Generation

A client can generate a third-party request by using the `CThirdPartyRequest` class provided in `MOOSGenLib`. The class's methods allow specification of the task type, job name. For every line that one would usually see in a Task configuration block one call to `CThirdPartyRequest::AddProperty()` is made specifying the property name and value. This done, a call to the `ExecuteTask` function returns a string that can be transmitted via the `CMOOSCommClient` or through a call to `MOOSApp::Notify()` under the name of `THIRDPARTY_REQUEST`. The string returned by the `ExecuteTask` function simply packs all relevant information in a string in a manner that is understood by the Helm.

15.4 Dynamic Controllers

We shall now discuss how each task maps navigation data and task goals to desired actuation settings.

Each task possesses two standard PID controllers (Proportional Integral Differential)– one for yaw control and one for depth control. All tasks that use navigation data as feedback for motion control use one or both of these controllers. The yaw axis controller is simple and is encapsulated in the class `CScalarPID` found in `MOOSGenLib`. The output of the controller is the tasks "vote" on `DESIRED_RUDDER`.

15.4.1 Scalar PID

The scalar PID has the structure shown in figure 15.4.1. It has integral limits to prevent integral wind up and is also output limited to bound the output values. Hence three gains and two limits specify the characteristics of the controller.

It is important to note that the controller does not require that it be run a precisely regular time intervals. Instead it uses the sequence of time stamps on the input navigation data to perform the differentiation and integration as required. The differentiation is performed using a 5-tap FIR filter. This leads to smooth performance. However no phase advance is performed – a point that could be improved upon.

15.4.2 Vertical Control via Pitch Control

The control of motion in the vertical direction is only applicable to the subsea case. Here we choose to control depth via pitch. The overall controller is fourth order and consists of an inner scalar PID loop controlling vehicle pitch. This loop maps error in

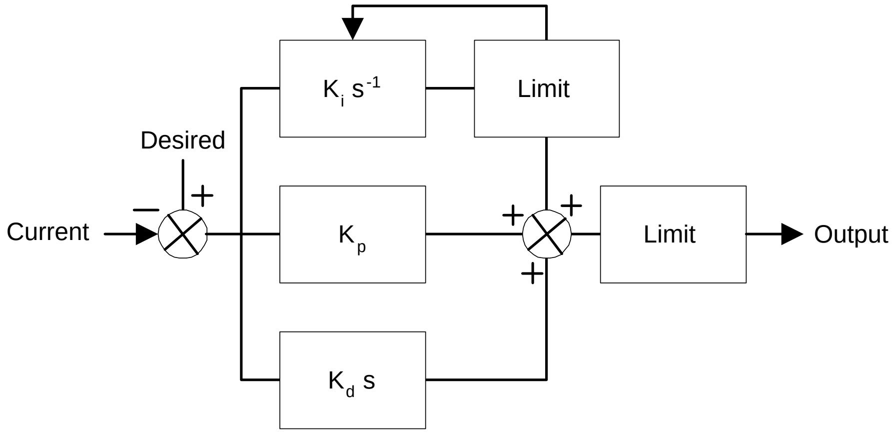

Figure 8: The structure of the scalar PID

pitch to desired elevator. The set point for the pitch loop is derived from the error in the vertical direction [4]. This topology is shown in figure 15.4.2.

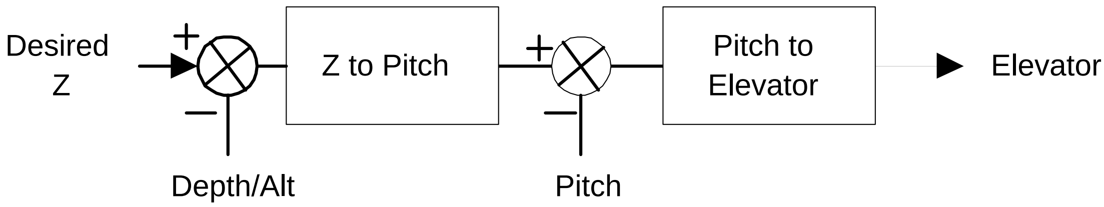

Figure 9: The structure of Z axis dynamic controller

15.4.3 Track-line Control

The `TrackLine` task is an interesting case study of the use of the dynamic controllers. It is derived from the `WayPoint` task which uses yaw control to drive in a straight line to a goal position. The `TrackLine` is more sophisticated in that it draws the vehicle onto a line defined between a start and goal point and then heads for the goal point along the track-line direction. Beneath the hood the task is continually changing the goal coordinates of the underlying way-point task – a "carrot and donkey" approach. At each time step the goal coordinate is set to be the projection of the vehicle location onto the track-line plus some "lead distance" along it.

15.4.4 The Independence Assumption

Implicit in all this is that the dynamics in the XY plane and Z axis are decoupled. Clearly this assumption may be too strong for some vehicles[5].In this case XY controlling tasks

[4]note altitude control and depth control have sign inversion between them. Depth is in the negative Z direction

[5]Although two AUV's have been succesfully controller using this strategy.

will also need to place a vote on elevator as well as rudder commands. This is no big problem, but they will need to be extended to control in the vertical plane at the same time. In other words such a vehicle will not be able to tolerate the division of tasks into XY and vertical control types as is currently the case.

16 Navigation – pNav

The pNav process is perhaps the most complex of all the MOOS processes. Its job requirement is simple –

> *"taking asynchronous, anachronistic and inconsistent input from sensors, provide a single , up-to-date estimate of vehicle pose and velocity."*.

The most important output of `pNav` is the `NAV_*` (said "Nav Star") family where the wild card is any one of *X, Y, Z, DEPTH, PITCH, ROLL, YAW, SPEED, X_VEL, Y_VEL, Z_VEL or YAW_VEL*. Note that velocities are in earth coordinates not body coordinates. The `NAV_*` family should be taken to be the best estimate of vehicle state. Indeed the Helm application uses this family to deduce actuation commands.

16.1 Priority Queues

pNav allows the route by which `NAV_*` information is derived to be specified in it control block. This is achieved by the use of Priority Queues or Stacks. The idea is simple – each `NAV_*` output has its own line in the configuration block. Reading from left to right specifies the preferences for deriving state estimation. For example, it might be preferable to have an AUV use pure GPS fixes for X,Y navigation when on the surface. In this case the GPS_X and GPS_Y sensor outputs are mapped straight through to NAV_X and NAV_Y and the first entry on the RHS of the "X" line in the pNav configuration block would be "GPS@TIMEOUT". Here the value of "TIMEOUT" would be the maximum time between GPS fixes that could be tolerated before assuming the sensor has stopped publishing valid data. For example if no GPS_X data arrived for TIMEOUT seconds then the GPS sensor would no longer be fed through to NAV_X and the navigator would shift attention to the next "SOURCE@TIMEOUT" pair in the priority queue. A likely second entry would be `DVL`. When the vehicle dives GPS fixes will cease and it may be desirable to dead-reckon using a DVL in its place. In this case the second entry would be "DVL@TIMEOUT". As soon as the vehicle surfaces and GPS fixes resume the stack/queue would "pop" and NAV_X would once more derive itself from GPS data. Figure 16.1 gives an example definition of navigation priority queues.

Of course this scenario assumes that the DVL sensor can produce DVL_X/Y measurements. It is more likely to produce reliable body velocity measurements. Now we must ask how to derive position estimates from sensors that do not measure positions? The `pNav` process solves this problem by providing two kinds of real time navigation filters: a ten state non-linear Kalman filter and a five state non-linear Least Squares filter. Collectively these filters provide the following functionality:

- Derivation of rate from position sensors
- Derivation of pose and rate from LBL ranges
- Optimal fusion of rate and position measurements
- Estimation of tide-height

```
////////////////////////////////////
//   routing priority stack
////////////////////////////////////
X      = GPS @ 2.0, EKF @ 2.5 ,LSQ @ 5.0,
Y      = GPS @ 2.0, EKF @2.0 ,LSQ @ 5.0,
Z      = EKF @ 2.0 ,LSQ @ 5.0,
Depth = EKF @ 2.0 ,LSQ @ 5.0,DEPTH @ 1.0
Yaw    = EKF @ 2.0 ,LSQ @ 5.0, INS @ 1.0
Pitch  = INS Speed = EKF
```

Figure 10: Specifying NAV_* derivations. Here for example NAV_X is derived preferentially from GPS (ie straight from GPS_X) unless no GPS data is received for 2 seconds. In this case EKF_X is used in its place. If the EKF (extended Kalman filter) fails to produce any output for 2.5 seconds then as a last resort LSQ_X is used (non-linear least squares). However as soon as GPS data begins to be delivered once more the system switches back to using GPS – popping the stack.

- Automatic robust rejection of outliers
- Self diagnostics

These navigation filters reside in the MOOSNavLib library, which is linked with pNav. They are discussed in greater detail in section 16.2.

16.2 Stochastic Navigation Filters

The navigation filters are moderately complicated software components and require a good understanding of stochastic estimation - the full extent of which is beyond the remit of this paper.

There are two kinds of navigation filters used for online navigation: a non-linear Kalman Filter (EKF) and a non-linear Least Squares Filter (LSQ). [6]

Least Squares Filter(LSQ) The LSQ filter is a conventional non-linear least squares filter using the Newton-Raphson iteration scheme. It can only produce estimates of pose (no rate states) and process measurements that are proportional to pose (no velocity measurements). Internally measurements are accrued (up to a certain time span) until the state space becomes observable, at which point a solution is found iteratively by successive linearizations. The accumulation of observations then begins again until the next estimate can be formed. Heuristically speaking the LSQ filter "starts from scratch" after every published estimate and has no concept of the flow of time across successive estimates. This implies that it is not easy to use measurements of rate in deriving pose estimates. On the other hand, it does mean the LSQ filter is robust and immune to errors in previous estimates.

[6] The Top down calibration filter within MOOS is used for automatic calibration of an LBL net using GPS and acoustic ranges and is not considered to be online.

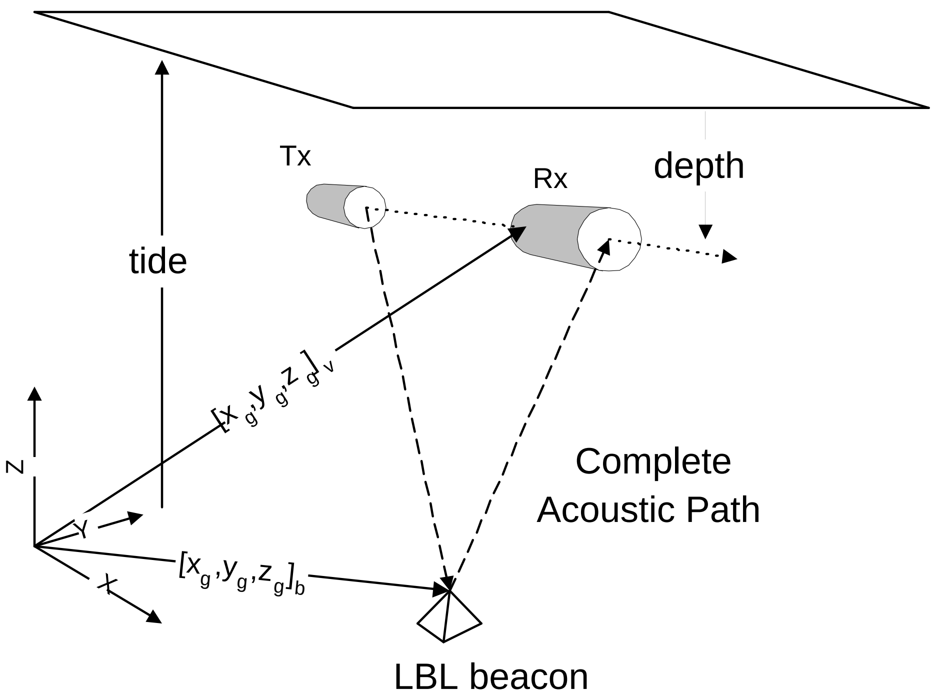

Figure 11: The relationship between tide, Z and depth. The navigation filters understand LBL time of flights as a function of two vehicle positions (to be estimated) and the beacons location (known).

Extended Kalman Filter (EKF) The EKF is the most complex of the two filters and also is the most versatile and accurate. It fuses predictions derived from a simple vehicle model with measurements supplied by sensors. Internally it use a fine discretization of time to faithfully interpret and model sensor measurements. The filter is recursive and so, heuristically speaking, possesses a concept of time. This enables it to produce estimates of velocity in addition to pose using pose only measurements.

The filters are designed to be fully configurable in an intuitive way from text in the `pNav` configuration block in the mission file. A typical `pNav` block is given in the Appendix. This section will use this listing as an example and discuss the meanings and use of some important constructs within it.

16.2.1 Defining Sensors

The filter needs to know what sensors are in use and their geometry with respect to the vehicle's coordinate frame – this is achieved with the `SENSOR_*` keyword where "*" is any of XY, LBL, ORIENTATION, DEPTH, BODY_VEL or ALTITUDE. The general syntax is as follows:

$$\texttt{SENSOR\_*} = \textit{iSource} \rightarrow \textit{sensorname} \ @ \ \textit{x,y,z,[twist]} \quad \textit{noise}$$

Anthropomorphically the syntax can be read as:

> Declare a sensor of type (*) which is managed by a process called *iSource*. Call it *sensorname* and understand it to be located at the vector (x, y, z) in body coordinates with a twist of *twist* about the body z axis (useful for compass offsets). Expect the sensor to produce data corrupted by noise with standard deviation *noise*.

Note there is no limit on the number of sensors that can be used. For example you can have two GPS sensors (XY). In fact on a large vessel this will allow estimation of yaw without an orientation sensor - for "free".

16.2.2 Mobile and Static Vehicles

The EKF allows the vehicle to be defined as static or mobile. A mobile vehicle includes velocity (x,y,z and yaw) estimates in its state estimate. A static vehicle has only pose estimates. The LSQ filter implicitly uses a static vehicle model as it has no concept of prior history and recalculates fixes from scratch whenever possible.

16.2.3 Defining Vehicle Dynamics

The EKF uses a linear, constant velocity dynamic model for the vehicle. It assumes that the vehicle will continue to move in a straight line with constant velocity across time steps. There is however a noise model associated with this model that dilutes the precision of the state estimate over time (in the absence of data from sensors). The degree to which this happens is governed by the `EKF_*_DYNAMICS` variables, which vary from 0 to 10. A setting of zero implies the vehicle is immutable in that direction and never changes. For example, a planet sized vehicle might have a setting of zero for its yaw dynamics. A setting of 10 implies that the vehicle is very mobile in a given degree of freedom and in the absence of observations from sensors over a few seconds we expect to have a large uncertainty in our state estimate. Essentially these numbers are parameterizing the expected error in our state model. Heuristically a larger number means that we are not so sure the vehicle always obey a constant velocity model.

Note that we do *not* inflict the constraint that the vehicle translates along the direction it is pointing (although when moving at speed this may be the case). Intentionally the vehicle model does not couple angular and cartesian degrees of freedom. Of course that is not to say that models explaining sensor data do the same - in particular the use of DVL body velocity observations correlates yaw and position estimates.

16.2.4 Start Conditions

The LSQ filter can be used to initialize the EKF. If however the EKF is being run in stand-alone mode it needs an initial guess to start up. The initial state guess is provided in the mission file via the `EKF_*` and the uncertainty (1 standard deviation bound) by the `EKF_SIGMA_*` variables. If the vehicle is type "mobile", velocity estimates are internally initialized to zero with a suitable uncertainty.

16.2.5 Logging

The performance of the navigation filters can be logged to file for post-mission analysis[7]. The location and stem name of these log files can be specified using the `NAV_LOG_*` variables. If time stamping is required the file name includes the creation time of the log in standard MOOS format. Two kinds of files are created: **.olog* and **.xlog*. The former logs all observations presented to the filter and the outcome of their processing - rejection/acceptance etc. The latter logs the evolution of the state estimate and its covariance. Both files are in text format and intended for simple quick parsing in Matlab.

16.2.6 Data Rejection

Both the EKF and the LSQ filters possess moderately sophisticated machinery, which allows them to discriminate between good and out of bound (outliers) sensor data.

The LSQ filter builds a temporal history of sensor data and applies some robust statistics tests to it to identify outliers. The underlying assumption here is that sensor noise is high frequency whereas the vehicle motion and hence reliable sensor data are low frequency processes. The `CSensorChannel` class tries to find the best (in terms of consensus) linear relationship between recent sensor data points. This done, it is able to identify outliers and remove them from the input stream feeding into the filter. The `LSQ_REJECTION` entries specify sensors (and acoustic channel for the case of LBL) which should be piped through this process. Each such statement declares the number of data points to be analyzed (the extent of the temporal history) and tolerable/expected deviation of point from the fitted line. For the case of LBL sensors this is a time measured in seconds.

The EKF filter is a little smarter and uses its current state estimate and uncertainty to identify that the current crop of observations are not mutually consistent either with themselves or with the last state estimate (a common technique in Kalman filtering). However having detected that at least one of the current sensor measurement makes no sense it employs geometric projective techniques to identify the true outlier.

16.2.7 EKF Lag

The EKF can be told to run "behind time". That is, only process sensor data up until η seconds ago. Having done this the EKF then forward predicts to the current time to produce an external navigation estimate (which is used to make dynamic control decisions). The next iteration will use the "un-altered" internal estimate as a prior (ie the η second old one). One might ask "why bother?". The reason stems from the fact that sensors cannot (should not) be relied upon to produce up to date measurements. Typically they are always out of date by the time they are processed. Things are exacerbated in the case of LBL data processing. Envision the case where an LBL transceiver ranges to two beacons, one of which is close, say 150 m away, with a reply delay of 0.1 seconds, and the other one is distant, say 1.5km away with a reply delay of 1

[7] The `MOOSMat` matlab script is designed to do this graphically.

second. The reply from the first beacon will come in 0.3 seconds after the interrogation but the remaining beacon's reply will not be heard for three seconds. Only when all replies are in (or the receive window has timed out) will the time of flight be transmitted by the sensor hardware. This means that by the time the data is processed it is at least 3 seconds old. Now, to properly "explain" these time measurements the filter must use the state estimates of the vehicle valid at the time-stamp of the data - in this case three seconds ago. Hence the filter needs to run behind time by an amount specified by the `EKF_LAG` parameter. The filter buffers all sensor data in a queue (which is self-sorting on time stamp) and extracts data from it valid at the current time minus whatever the lag is set to.

16.2.8 Fixed Observations

It is possible to force a constraint on the navigation by declaring persistent observations. Every time the filter "ticks" these artificial observations are added to a list observations stemming from real sensors. This for example would allow a surface craft with no depth sensor to use LBL navigation by declaring a zero depth artificial observation. Another case is when a heading measuring device is not available but a constant heading observation can be added to take its place. (Recall that the vehicle model does not correlate heading and position so with a constant heading observation the vehicle will simply translate).

16.2.9 LSQ/EKF Interaction

If both the LSQ and EKF filters are run at the same time the LSQ filter can be used to initialize and monitor the performance of the EKF. The fact that although they have lower precision LSQ estimates have no dependency on prior estimates can be exploited. This can be advantageous for several reasons:

Robustness A poorly tuned/setup EKF can diverge - that is, produce a state estimate so far from the truth and with sufficient confidence that it rejects all incoming sensor data. There is no reason that this should happen given the data rejection schemes employed, however navigation is stochastic and unlikely things will happen given time. A deployed system should be robust to such occurrences and the checking of estimates against the LSQ filter offers one (of several employed) way to do this. If the estimates emerging from the LSQ differ significantly from those emerging from the EKF over a sustained interval `pNav` takes the EKF off-line and attempts to re-initialize it to the LSQ estimates. We require a lack of consensus over several seconds/fixes to prevent spurious LSQ fixes pulling the plug on the EKF prematurely. This also accords with the observation that once the EKF has diverged it is unlikely to recover and so waiting a few seconds is not going to change anything.

Observability The LSQ estimate essentially solves a nonlinear set of equations to derive state observation data. No state estimate can be derived until the system of equations is observable - ie enough of the right kind of measurements have been accrued. If no LSQ solution can be produced for an extended period it provides *prima-facie* evidence that the navigation problem is or has become(via sensor failure) unobservable and unsolvable. Note that the EKF does not require full observability to operate - unobservable states just become more uncertain with each iteration[8]. The navigation control block allows a LSQ timeout to be specified. If no LSQ fixes are derived for a time exceeding this an navigation failure is declared and all filters are taken off-line. This mechanism has two applications. Firstly it prevents an autonomous mission from being started on a vehicle that cannot be navigated. Secondly, it serves to monitor sensor failure and prevent motion control decisions being taken on the basis of at least partially "predict only" EKF navigation estimates.

Automatic Initialization The LSQ does not need a prior to produce a state estimate. Hence it does not need to be primed with an initial guess. Hence once a stable/repeatable stream of solutions begins to flow from the LSQ filter they can be used to initialize the EKF removing the need for approximate starting conditions to be specified.

16.2.10 Hidden State Estimation

Both the EKF and the LSQ estimate the tide height - the distance between the XY navigation plane and the surface. The EKF can optionally also estimate a bias term in an orientation sensor (one bias is applied to all orientation sensors). Great care has to be taken in ensuring that this term is in fact observable given the system configuration - heading bias estimation should only be enabled when operating with a body velocity sensor *and* a position sensor.

16.2.11 Navigation Failure

Limits can be placed on the permissible navigation uncertainty. If the one sigma bound of the estimates exceed these limits a *Navigation Failure* is declared. All relevant filters are taken off-line and the `EndMission` variable is published. This is the last line of defence against navigation failure - if this happens things have gone badly wrong and the mission should be terminated. The EKF also watches the numerical value of the pose derivative states. If they exceed sensible limits (10m/s or $\pi rads^{-1}$) the states are reset to zero. This is essentially a coordinate shift and so is statistically consistent although somewhat alarming. Accordingly a warning is issued. This condition is often experienced at boot time when a large velocity is inferred to explain the apparent shift in position from initial guess to that of the first EKF derived estimate.

[8]Until specified limits are reached at which point the filter is taken off-line.

16.3 Sensible Configuration and Commissioning

This section is intended as a brief guide to actually using the navigation filters and owning process `pNav`. It is not exhaustive and cannot replace experience and understanding in the underlying techniques.

16.3.1 Defining the Navigation Frame

Different groups of people like to define coordinate origins in different places. In particular the $Z = 0$ plane. In an area with little or no tidal flux the use of an artificial tide observation (tide = 0) can be used to fix the $Z = 0$ plane to the sea surface. All beacon locations will need to have negative Z coordinates[9].

16.3.2 Heuristic Hints

The following are a selection of heuristic statements that are helpful to keep in mind during commissioning and verifying the navigation component of MOOS.

- If rate states are often being reset something is wrong. Poor data is being accepted when it should not be.
- LSQ filter cannot use velocity sensors (it has no idea of history).
- The LSQ filter cannot be used underwater without an LBL net – there are no sensors that measure a quantity proportional to position.
- Increasing vehicle dynamics settings will cause more observations to be accepted but reduce resilience to bad sensor data.
- Decreasing vehicle dynamics will lead to smoother (piece-wise) trajectories but may cause sensor data to be erroneously rejected during swift manoeuvres.
- Increasing estimated sensor precision (in the sensor declaration) will tend to cause more data to be accepted and increase its effect on state estimates. Accepting bad data will increase the chances of the state vector being corrupted to the degree that no data is ever accepted again.
- Try to keep the `EKF_LAG` setting as small as possible. If set too small LBL data will not be used. If set too large the final prediction step used to bring the estimate forward to the current time will be inaccurate - bad news for dynamic control.
- The `EKF_LAG` can be very small if no LBL sensing is used.
- Tide estimation can only be accomplished if both depth sensors and LBL data is used (with beacons on the sea bed).

[9]However depth will still be positive as depth =tide-Z.

- Tide estimation cannot be used using only altitude and LBL measurements as such a scheme would require a model of the sea-bed to explain the altitude data.
- **Always** analyze the navigation logs before performing a long mission with a new sensor configuration or LBL array. Do not launch unless everything seems fine. There are numerous safety features built in but it could be several minutes before problems are detected and the navigator pulls the plug and emits the navigation failure flag (which should be monitored by the `EndMission` task).
- If the EKF is being run by itself (ie not booted from the LSQ) make sure that its start coordinates and uncertainties are suitable to allow it to start accepting data when it starts up – ie close enough to its true position!
- It is a good plan to set the initial uncertainty in heading to be large. Otherwise orientation data may not be accepted.

17 Manual Control - `iRemote`

`iRemote` was designed to be a control terminal for a deployed vehicle. It is really nothing more than a long `switch` statement based on characters input from the keyboard. One of its many functions is to allow remote control of the actuators of the vehicle. This is an invaluable asset for land and sub-sea vehicles alike. The application is multithreaded. The primary thread blocks on a read of keyboard input. When a character is pressed some action is taken - for example publishing a new value for `DESIREDTHRUST`. The fact that `iRemote` can take control of a real vehicle presents a safety problem. What if the human controller walks away or even worse the vehicle moves out of communication range (eg a submarine dives) and the console is not available? To prevent the last issued actuator command being carried out *ad-infinitum* a secondary in thread `iRemote` prompts the user to hit an acknowledge key (') at least every 15 seconds. If the human driver does not respond the all actuators are set to the zero position.[10]

17.1 Summary of Functionality

The following (not exhaustive) list describes some of the online functionality that `iRemote` provides:

[10] The actuation driver in iActuation will also shut down all motors if it does not receive control commands for an extended period of time. In the Bluefin vehicle the driver class within iActuation sends a message to the Janitor processes which resets a watchdog on the power management board - keeping the tail cone powered.

Function	Key	Comment
Reload Mission File	R	Tells `pHelm` to rebuild its tasks
Restart Navigation	V	Tells `pNav` to reboot all navigation filters
Restart Logger	G	Tells `pLogger` to begin recording to a new set of log files
Begin Mission	O	Instructs `pHelm` to go online
Halt Mission	O or *space*	`pHelm` goes offline and `iRemote` takes control immediately.
Navigation Summary	*	Prints a summary of salient navigation information
Rudder Left/Right	N,M	Steer control
Elevator Up/Down	P,L	Pitch control
Thrust Up/Down	A,Z	Throttle control (+ shift gives 100 percent)
Stop	*space*	Immediate zero of all degrees of freedom
Fetch DB	F	Prints a summary of the contents of the entire `MOOSDB`
CustomKey	[0 → 9]	The numeric keys can be made (via `iRemotes` configuration block) to publish any named variable with a specified value.
CustomSummary	+	The configuration block allows a custom summary to be built consisting of any variable names used within the system. `iRemote` subscribes to this data and prints its current value when requested.

17.2 Informing the Pilot

In most missions `iRemote` is the only interface the vehicle pilot has with the vehicle. Clearly then a method is needed by which *important* information can be sent to the `iRemote` console from any process. The `CMOOSApp` member function `MOOSDebugWrite` achieves this by issuing a notification on a variable watched by `iRemote`. Such messages are displayed on the iRemote console at run time along with the process making the announcement. Note that this name is somewhat unfortunate as this function should not be used for debugging - it is a run-time thing. It is frustrating to have a cornucopia of messages flashing on the screen during a mission the content of which is meaningless to the pilot. Typical uses of this functionality would be a very occasional summary of navigation status and system level warning messages - for example notification of unexpected mission task termination.

18 Logging - pLogger

The `pLogger` process is intended to record the activities of a MOOS session. It can be configured to record a fraction of or every publication of any number of MOOS variables. It is an essential MOOS tool and is worth its weight in gold in terms of post-mission analysis, data gathering and post-mission replay.

The configuration of `pLogger` is trivial and consists of multiple lines with the following syntax:

$$\texttt{Log} = \textit{varname} \ @ \ \textit{period} \ [NOSYNC]$$

where *varname* is any MOOS variable name and *period* is the minimum interval between log entries that will be recorded for the given variable. For example if *varname*=`INS_YAW` and *period* = 0.2 then even if the variable is published at 20Hz it will only be recorded at 5Hz. The optional `NOSYNC` flag indicates that this variable should not be recorded in the synchronous logs (see section 18.1)

18.1 Log File Types

The logger records data in two file formats - synchronous ("slog" extensions) and asynchronous ("alog" extensions). Both formats are ASCII text – they can always be compressed later and usability is more important than disk space. The two formats are now discussed.

18.1.1 Synchronous Log Files

Synchronous logging makes a table of *numerical* data. Each line in the file corresponds to a single time interval. Each column of the table represents the broad evolution of a given variable over time. The time between lines (and whether synchronous logging is even required) is specified with the line

$$\texttt{SyncLog}= \textit{true/false} \ @ \ \textit{period}$$

where *period* is the interval time.

If there has been no change in the numeric variable between successive time steps then its value is written as `NaN`. It is important to note that synchronous logs do not capture all that happens - they sample it. Synchronous logs are designed to be used to swiftly appraise the behaviour of a MOOS community by examining numeric data in a tool such as Matlab or a spreadsheet. The `MOOSData` Matlab script reads in these files and with a single mouse click can display the time evolution of any logged variable.

18.1.2 Asynchronous Log Files

Asynchronous logging is thorough. The mechanism is designed to be able to record *every* delta to the MOOSDB. The use of the period variable allows the mission designer to back off from this ultimate limit and record variables at a maximum frequency. The key properties of asynchronous can be enumerated as follows:

1. Records both string and numeric data
2. Records data in a list format - one notification per line

3. Entries only made when variable is written

Asynchronous log files are designed to be used with a playback tool (for example `uPlayback` or other purpose-built executable). Although the handling of strings and numeric data adds a slight overhead to such a program's complexity the utility gain from being able to slow, stop and accelerate time during a post-mission replay/reprocessing session is simply massive.

18.1.3 Mission Backup

Simply having the *alog* and *slog* files is not enough to evaluate the mission. One also needs the things that *caused* the data to be recorded, namely the *.moos Mission file and the *.hoof file (if Task redirection was used). To this end the `pLogger` process takes a copy of these files and places them (name appended with a time stamp if desired) within the logging directory. The files extensions are renamed to **._moos* and **._hoof* respectively.

18.2 Replay – `uPlayBack`

There is a Windows-based GUI application that can load in *alog* files and replay them into a MOOS community as though the originators of the data were really running and issuing notifications. A typical use of this application is to "fake" the presence of sensor processes when reprocessing sensor data and tuning navigation filters. Alternatively it can be used in pure replay mode perhaps to render a movie of the recorded mission. The GUI allows the selection of which processes are "faked". Only data recorded from those applications will be replayed from the log files. Although the GUI off this utility application is not platform independent (MOOS only guarantees online platform independence) the heart of it is. There is a single class that encapsulates all the replay functionality - `CMOOSPlayback`. The GUI simply hooks into the methods exported by this class.

A client process can control the replay of MOOS messages by writing to the `PLAYBACK_CHOKE` variable add writing a valid time in the numeric message field. The Playback executable will not play more than a few seconds past this value before waiting for a new value to be written. In this way it is possible to debug (halt inspect and compile-in-place etc) at source level a client application using replayed data without having the playback rush on ahead during periods of thought or code-stepping.

19 Startup - `pAntler`

The process `pAntler` is used to launch/create a MOOS community. It is very simple and very useful. It reads from its configuration block a list of process names that will constitute the MOOS community. Each process to be launched is specfied with a line with the syntax

`Run` = *procname* [@ `NewConsole` = *true/false*]

The optional console parameter specifies whether the named process should be launched in a new window (an xterm in Unix or cmd-prompt in NT derived platforms). Each process launched is passed the mission file name as a command line argument. When the processes have been launched `pAntler` waits for all of the community to exit and then quits itself.

`pAntler` provides a simple and compact way to start a MOOS session. For example if the desired mission file is *Mission.moos* then executing

```
pAntler Mission.moos
```

will launch the required processes for the mission. Of course a sensibly designed mission will not actually start to do anything until a human (via `iRemote`) has confirmed a good working status of the processes involved (eg `pNav`) and actively hands control over to the Helm.

19.1 I/O Redirection - Deployment

As already mentioned, frequently `iRemote`, displayed on a remote machine, will be the only interface a mission pilot has to the MOOS community. We must ask the question - "where does all the IO from other processes go to prevent I/O blocking?". One answer to this is I/O redirection and backgrounding MOOS processes - a simple task in unix derived systems [11]/

Running `pAntler` in the following fashion followed by a manual start up of `iRemote` is the recommended way of running MOOS in the field using a serial port login.

1. `pAntler` *mission.moos* > ptyZ0 > /dev/null &

2. `iRemote` *mission.moos*

This redirection of `iRemote` is encapsulated in the `moosbg` script included with the MOOS installations. In the case of an AUV the interface can only be reached through in-air wireless communications, which will clearly disappear when the vehicle submerges but will gracefully re-connect when surfacing(not so easy to do with a PPP or similar link).

20 Visual Debugging - `MOOSScope`

The `MOOSScope` is another GUI application. It allows a user to place a stethoscope on the MOOS network and watch what variables are being written, which processes are writing them and how often this is happening. After starting up the scope and specifying the host name and port number of the `MOOSDB`the user is presented with a list of all MOOS variable in the server and their current state. Several times a second `MOOSScope` calls into the DB and uses a special/unusual (and intentionally undocumented) message that

[11]some OS are good for development others for running...

requests that the server inform the client about *all* variables currently stored along with their update statistics. `MOOSScope` is a central tool in the MOOS suite. It can be run from any Windows (once again utilities are not platform independent in this release) machine to spy on and present visual feedback on any MOOS community.

20.1 Poking the MOOS

`MOOSScope` has one other valuable use : poking the MOOS. It allows a user to double click on a variable name and alter its value (string or double) interactively. This is akin to changing memory contents in a source code debugging session. The difference here is that this action is a notification and all clients that are registered for it receive a message in their mail box and act on it accordingly. The utility of this functionality should not be underestimated. For example, during the commissioning of a new sensor (say a DVL) it may be unclear what the best configuration parameters are. For example by having the managing process subscribe for notifications on a `PARAMETERS` variable the `MOOSScope` can be used to rapidly explore the performance/parameter space by simply poking new configuration describing strings into the `PARAMETERS` variable.

21 Conclusions and Further Work

This paper provides a discussion of the MOOS software. It is however, by no means exhaustive. Although the architecture of the core communications are completely described many of the details of the higher level processes building on this architecture have been omitted. In some ways this is in the spirit of the MOOS - knowledge of the innards of individual processes can be administered on a "need to know basis". What is important is the interactions between processes. In a similar vein the details of of the workings navigation filters have also been omitted although their broad functionality is described.

It is hoped that MOOS can provide a flexible research tool for some time to come. It is an evolving project and under continual improvement and expansion. For example the near future should see its application to multiple vehicle scenarios in both land and sub-sea domains.

Most of all it should be fun to use - despite its ridiculous name.

Appendix A - Skeleton Code

```
#include "MOOSLib.h"
#include "MOOSGenLib.h"
#include "Skeleton.h"
int main(int argc ,char * argv []) {
    // set default mission file
    char * sMissionFile="Mission.moos";

    if (argc>1)
    {
        sMissionFile = argv [1];
    }

    //make your application
    CSkeleton TheSkeleton;

    //run it//
    TheExample.Run("Skeleton",sMissionFile);

    return 0;
}
```

Figure 12: source code for a very minimal application that runs a `CMOOSApp` derived object called `TheSkeleton` of type `CSkeleton`.

```
class CSkeleton : public CMOOSApp
{
 protected:
    //these are the virtual functions to overide
    bool OnNewMail(MOOSMSG_LIST &NewMail);
    bool Iterate ();
    bool OnConnectToServer();
    bool OnStartUp();
};
```

Figure 13: source code for the declaration of the example `CSkeleton` class

```
#include "Skeleton.h"

bool CSkeleton::OnNewMail(MOOSMSG_LIST &NewMail)
{
    //parse mail here
    return true;
}

bool CSkeleton::OnConnectToServer()
{
    //register for variables here ....
    return true;
}

bool CSkeleton::Iterate ()
{
    //do standard work here
    return true;
}

bool CSkeleton::OnStartUp()
{
    //do start up things here ...
    //for example read from mission file ...
    return true;
}
```

Figure 14: source code for the `CSkeleton` `MOOSApp` derived class

Appendix B - Navigation Configuration

```
////////////////////////////////////////////
// pNav control block
ProcessConfig = pNav {

    //////////////////////////////////////////
    //      routing priority stack
    //////////////////////////////////////////
    X           = GPS @ 2.0,EKF @ 2.0 ,LSQ @ 5.0,
    Y           = GPS @ 2.0,EKF @ 2.0 ,LSQ @ 5.0,
    Z           = EKF @ 2.0 ,LSQ @ 5.0,
    Depth       = EKF @ 2.0 ,LSQ @ 5.0, DEPTH @ 1.0
    Altitude    = RANGE @ 1.0
    Yaw         = EKF @ 2.0 ,LSQ @ 5.0, INS @ 1.0
    Pitch       = INS
    Speed       = EKF

    ALWAYS_READ = INS_YAW,PARA_DEPTH,LBL_TOF,GPS_X,
                  GPS_Y,DVL_BODY_VEL_Y,DVL_BODY_VEL_X,DESIRED_THRUST

    ////////////////////
    // FILTER CONTROL:
    ////////////////////

    //what filters to use ......
    UseLSQ      = true
    UseEKF      = true
    LSQTimeOut = 180

    MaxLSQEKFDeviation = 20
    MaxEKFPositionUncertainty = 30

    //how to log nav if required (development tool)
    NAV_LOG = true
    NAV_LOG_PATH = C:\codescratch\MOOSLOG\
    NAV_LOG_TIMESTAMP = false


    SV = 1500.0

    //////////////////////////////
    // static observations
    // format is  [Value] @ [uncertainty]
    //////////////////////////////

    FixedTide       = 0 @ 0.1
```

```
    ////////////////////////////////
    // map sensor outputs to sensors within filters
    // specifying geometry .....
    // Syntax is as follows
    // SensorType       =Source -> SensorName @ Location ~ Sensor Std
    //////////////////////////////

    SENSOR_XY    =   iGPS    -> TheGPS  @ 0,0,0 ~ 0.3
    SENSOR_LBL   =   iLBL    -> TheAvtrak @ 0,0,0 ~ 0.009
    SENSOR_ORIENTATION = iINS -> TheINS @ 0,0,0 ~ 0.01
    SENSOR_DEPTH   =   iDepth  -> TheDepth @ 0,0,0 ~ 0.1
    SENSOR_BODY_VEL = iDVL ->  TheDVL  @ 0,0,0 ~ 0.05


    ///////////////////////////////
    // Least Square filter set up
    LSQ_REJECTION = TheAvtrak[1] : History = 6 ,FAIL = 0.005
    LSQ_REJECTION = TheAvtrak[2] : History = 6 ,FAIL = 0.005
    //etc ...
    //etc ...

    ///////////////////////////////
    // Extended Kalman Filter set up

    EKF_LAG =3.0
    EKF_TIDE = 00

    //how fast/mobile is the vehicle?
    // 1=slow
    // 10 =fast
    EKF_XY_DYNAMICS = 0.1
    EKF_Z_DYNAMICS = 0.1
    EKF_YAW_DYNAMICS = 0.1
    EKF_VEHICLE_TYPE = MOBILE

    // initial uncertainty
    EKF_SIGMA_XX = 20.5
    EKF_SIGMA_YY = 20.5
    EKF_SIGMA_ZZ = 1
    EKF_SIGMA_HH = 180
    EKF_SIGMA_TIDE = 0.1

    // initial state
    EKF_X = 0
    EKF_Y = 0
    EKF_Z = 0
    EKF_H = 0
}
```

Appendix C - Coding Style Guide

This is the recommended coding style for c++ in MOOS. The author has tried to follow such rules throughout - of course there will be occasional slips and these should be corrected where found - its all about entropy[12]. The author urges future "MOOSies" to use the same style. It is not so much about one style being better than another but keeping things unified and having the same flavor whenever possible.

- Use Hungarian notation prefixes for variables"
  - doubles prefixed by "df" eg "dfMyLocalVal"
  - ints prefixed by "n"
  - strings prefixed by "s"
  - member variables begin with "m_" eg "m_dfMyMemberVal"
- Variables should be long enough to be meaningful. Where they are multi word use capital to denote first letter of each word. e.g. m_dfTurnAroundTime.
- Use capital letters to identify defined constants (or the equivalent thereof)
- Indent / tab stops in steps of 4
- Opening and closing brackets at same tab stop as outer, enclosed lines to be indented.
- Minimize scope of local variables.
- Do not re-use a single local variable to express different concepts at different points in a routine. There is a risk that somebody doesn't notice that when he comes to modify code at a later date.
- Be liberal in use of white space to separate logical steps - for clarity.
- Source text should not overrun 100 characters to a line. You should be able to get that across a portrait style A4 paper page if you want to print it out.
- Keep to simple expressions in a statement. It will be easier to understand later.
- Restrict the executable code of a procedure to about an A4 page. If it stretches much beyond that you can probably find a block of code that expresses an identifiable concept to remove to a separate procedure.
- Avoid the use of multiple inheritance and friend classes.

[12]Thanks to Greg Walker for this list

- Never sit in an un-blocked loop waiting for an externally controlled condition to be met. It eats up processor time with no benefit.

- Avoid use of explicit numeric values in code. Assign symbolic name to the number so that the definition appears in one place only.

- Avoid "copy and paste" of anything more than tiny code fragments. It's an indication that a common procedure can be created which will service multiple needs. If there's less code in the application, there's less to go wrong. If it needs a fix then one fix solves all associated problems - you don't have to seek out the other similar code which probably has the same fault.

- Where procedure calls have lots of parameters, don't be afraid to assign a line to each parameter - but do put the opening bracket in the same text column as the closing bracket.

- TRACE statements should start with class::method identifier, and end with a newline. That makes it easy to find the source code that invoked the debug output. e.g. TRACE("CMyClass::DoIt - That's done it \ n");

Appendix D - Known Bugs and Inadequacies

The following is an incomplete list of the known issues with MOOS as of December 2002.

- Navigation filters do not take roll and pitch into account when processing DVL data - easy to fix as all sensor data is transformed into to a horizontal, rotated frame for processing.

- In Windows `iRemote`sometimes does not quit when told to. Strangely it does remove itself from the process table.

- The dynamic controllers do not take in rate information but rather infer it themselves which is a noise prone process and incurs a phase lag - easy to fix.