MOOS@LAMSS

Set up guide for the LAMSS[1] Software project

Principal author: T. Schneider (tes@mit.edu)
Started: 2.17.09, last modified January 10, 2017.

http://lamss.mit.edu
http://launchpad.net/lamss
http://moos-ivp.org
http://gobysoft.com

This document can be downloaded from http://gobysoft.com/dl/moos_at_lamss.pdf[2]

Contents

[1] laboratory for autonomous marine sensing systems

[2] The source for this document is in `lamss/src/doc/moos_at_lamss.tex` in the lp:lamss/lamss BZR repository.

Chapter 1

Motivation

The purpose of this document is to describe a number of official and *de facto* operating procedures for using the **M**ission **O**riented **O**perating **S**uite (MOOS[1]) within the context that we use it at the MIT **La**boratory for **M**arine **S**ensing **S**ystems (LAMSS[2]) under direction of Henrik Schmidt. A great deal of items in here are subject to change, as we find more efficient and user-friendly ways of doing them. I will attempt to keep this document up to date as much as possible, but please reference the date on the top and email the appropriate person with questions should reality conflict with the story in this paper. You are also welcome to commit your own fixes to this document directly. The LaTeX source is available in the bazaar repository `lp:lamss` in the folder `lamss/src/doc/moos_at_lamss`.

[1] http://www.robots.ox.ac.uk/~pnewman/TheMOOS/
[2] http://lamss.mit.edu

Chapter 2

Conventions

I will use the following conventions in this document:

- monotype font with > (e.g. `> command`) indicates a `command` you want to run on a terminal emulator prompt. Long commands are written in `bash` syntax so you can copy / paste into a shell.
- quotation marks (e.g. "colloquialism") indicates a synonym for a term you may hear, but isn't necessarily the official or preferred term.
- footnotes[1] give extra information or indicate a webpage where you can find out much more information.

[1] example footnote

Chapter 3

Code and compilation

3.1 Preparing your computer

You need a computer running a relatively new copy of Linux (Ubuntu[1] is highly recommended, as this is what I and the lab machines run). While MOOS itself compiles under Windows, the MIT/LAMSS software in general does **not**. From here I will assume you are using Ubuntu unless otherwise noted.

You will need to install a number of external libraries to be able to download and compile the MOOS and IvP source. If the package cannot be found, use (`> apt-cache search`) to find the version appropriate for your distribution version. Also, you can use (`> gksudo synaptic`) for a graphical way to install packages. For Ubuntu, you will need to install the following packages[2]:

```
# (bare bones)
sudo apt-get install \
     bzr \
     cmake \
     build-essential \
```

3.2 Getting code

We make use of a number of open source software projects. There are three that will concern you most directly:

[1] http://www.ubuntu.com

[2] the packages listed are for the newest version of Ubuntu. Often package names change slightly between releases. If you are having difficulty for example finding libxerces-c2-dev, do an search on the first part of the name for the version that is applicable (e.g. > `apt-cache search libxerces`)

1. The `moos-ivp` project[3], the code lives on the server `oceanai.mit.edu`, which is maintained primarily by Mike Benjamin (mikerb@csail.mit.edu). You can install a binary package of `moos-ivp` using `apt-get` (unless you need to modify parts of it).
2. The `goby` project[4], which lives on `launchpad.net`. You can install a binary package `goby` using `apt-get`.
3. The `lamss` project[5], the code lives on `launchpad.net` and the version control system used is `bazaar` (`bzr` for short).

bzr

This is not a tutorial on version control software: for that, reference http://doc.bazaar.canonical.com/en/. Print out http://doc.bazaar.canonical.com/bzr.dev/en/_static/en/bzr-en-quick-reference.pdf and put it on your wall. In a nutshell, bzr allows multiple people to share changes to a code base (or any files, for that matter) in an intelligent manner where users can be making changes concurrently. Each user has a local (working) copy of the repository (a "branch" in Bazaar) within which he makes changes. When the changes are to be shared to other users, he commits them (`> bzr commit -m "changes are so and so"`). When he wishes to update his working copy to incorporate changes made by other users, he does an update (`> bzr update`). If you have used Subversion in the past, it is syntactically similar to Bazaar. The main practical difference is that bazaar allows decentralized repositories. If you are new to version control you may find the `bzr explorer` tool useful. Do `> sudo apt-get install bzr-explorer` to install it and `> bzr explorer` to run.

Getting LAMSS code

For working on your laptop where you need to commit to repositories regularly, I recommend using the read/write checkout of these repositories (which requires SSH access through launchpad).

There are several code repositories, each with intended for a different level of "polish" and permissions. Testing and new (alpha) code belongs in `lamss-internal`, code that is shared with collaborators goes in `lamss-shared`, and publicly released code goes to `lamss`. There are also project specific repositories that typically depend on `lamss` and `lamss-shared`, and I will refer to all of these repositories collectively as `lamss*`.

The steps for checking out the code are:

- Sign up for a launchpad.net account: https://launchpad.net/+login, then email lamss@mit.edu your account name (upper right hand corner after you login) so we can give you permissions to the appropriate lamss team.

[3] http://moos-ivp.org
[4] http://launchpad.net/goby
[5] http://launchpad.net/lamss

- Give launchpad your RSA public key(s): https://help.launchpad.net/YourAccount/CreatingAnSSHKeyP
- Tell bazaar who you are on launchpad (substitute your id for "tes"):

```
bzr launchpad-login tes
```

- To checkout the code, use the following commands:

```
bzr co lp:lamss/lamss
# if you have permission
bzr co lp:lamss/lamss-shared
bzr co lp:lamss/lamss-internal
bzr co lp:lamss/lamss-{project1}
...
```

When these commands complete, you will have several folders at the same level in your directory structure (`lamss`, `lamss-shared`, and `lamss-internal`, for example).

3.3 Dependencies

After checking out the code, you should install the necessary dependencies. Since these dependencies are tied to the code itself, they are not listed in this document; a script is provided for your convenience (`dependencies.sh`), which will download and install these packages on an Ubuntu system. This script should be run with root user privileges, as follows:

```
> cd lamss
> sudo ./dependencies.sh
```

To install dependencies for building the documentation as well:

```
> sudo ./dependencies.sh doc
```

For non-Ubuntu users, check the package lists found in the `DEPENDENCIES` file. This file is actually a Makefile used by the install script, but the package names should be easy to find, listed under each `apt-get install` command.

3.4 Compiling

After checking out the code, installing dependencies, and after every subsequent update, you must compile the code to incorporate changes into the binaries (the files that are run by the operating system).

The dependencies between these repositories is strict:

```
lamss-internal
lamss-{project1} -> lamss-shared -> lamss
lamss-{project2}
```

Thus, you must **first** compile `lamss`, then `lamss-shared`, and finally `lamss-internal` and the various `lamss-{project}` repositories. Compile `MOOS-IvP` and the `lamss*` repositories by running:

```
cd ../lamss
./build.sh
```

The `build.sh` script that resides in `lamss`, `lamss-shared`, and `lamss-internal` will attempt to build all of these repositories in the appropriate order. Thus, you only need to run it once from inside one of these repositories.

If you're just getting started, you may want to skip to section 3.5. For more detail, keep reading.

Advanced details on configuring and building

We use `CMake` for generating Makefiles which are then processed by `make`.

You can pass command line parameters to the `configure.sh` scripts that are passed onto `cmake`[6]. If you wanted to compile with debugging symbols (for use with `gdb`, for example), you can run:

```
> ./configure.sh -DCMAKE_BUILD_TYPE=Debug
```

You can pass command line parameters to the `build.sh` scripts which are then passed to `make`. So, for example, to build `lamss*` using four jobs (good for multicore processors) and continue on errors (`> man make`), you could use

```
> ./build.sh -k -j4
```

It is useful to automate the process with the following shell script (make a file `update_compile_moos.sh` in the base folder as the repositories and copy the commands below; a variant on this script already exists in `missions-lamss/scripts`[7]):

update_compile_moos.sh:

[6] See documentation at http://www.cmake.org/cmake/help/documentation.html

[7] see https://bazaar.launchpad.net/~lamss-shared/lamss/missions-lamss/view/head:/scripts/update_compile_moos.sh

```
#!/bin/bash
# all command line parameters are passed to make
set -e -u

#where moos resides
path_base=~/

# updates
for repo in lamss*
do
  if [ -e ${repo} ]; then
    bzr update ${repo}
  fi
done

# compilation
pushd lamss
./build.sh $@
popd

popd

echo "success!"
```

Then, every time you need to compile simply type `> ./update_compile_moos.sh` in the folder where that script exists.

One final note: running `> cmake` only compiles the "default" settings for cmake (as determined by CMakeLists.txt files in every code directory. You may want to compile optional code at times. To do this, run `> ccmake` (note the extra 'c') instead of `> cmake` in one or more of the above locations. In newer versions of Ubuntu, you need to
`> sudo apt-get install cmake-curses-gui` to get access to `ccmake`.

In `ccmake` you can select various options to turn on and off. Type 'c' to configure after making changes (possibly several times) and then finally 'g' to generate the Makefiles, in the same way `> cmake` would.

iMatlab

iMatlab allows you to use MOOS from inside MATLAB. Since MATLAB is proprietary software and is not installed in a standard way, you will need to compile iMatlab separately from inside MATLAB.

If you want to build iMatlab use this command for compiling:

```
> sudo chown {your login name} /path/to/MATLAB/toolbox/local
> cd /usr/share/MOOS/iMatlab
> matlab < build_iMatlab.m
```

This should complete successfully and write iMatlab.mex* to `/path/to/MATLAB/toolbox/local`. Make sure that `/path/to/MATLAB/toolbox/local` is on your MATLAB path. On my system `/path/to/MATLAB/toolbox/local` happens to be `/usr/local/MATLAB/R2012b/toolbox/local`. This can vary depending on where MATLAB installed itself.

From MATLAB, do `> which iMatlab`. You should get: `/path/to/MATLAB/toolbox/local/iMatlab.mex*`

Depending on the version of MATLAB you're using, you may have an incorrect version of libstdc++ shipped with MATLAB for the compiler you're using. This manifests itself as an error like so:

```
Invalid MEX-file '/scratch/pkg/matlab/matlab-2009a/toolbox/local/iMatlab.mexa64':
/scratch/pkg/matlab/matlab-2009a/bin/glnxa64/../../sys/os/glnxa64/libstdc++.so.6: version
'GLIBCXX_3.4.11' not found (required by
/scratch/pkg/matlab/matlab-2009a/toolbox/local/iMatlab.mexa64)
```

The recommended solution to this is to move the MATLAB copy of the C++ library and then symlink that location to the system version:

```
cd /usr/local/MATLAB/R2012b/bin/glnxa64/../../sys/os/glnxa64
sudo mv libstdc++.so.6 libstdc++.so.6.matlab-backup
sudo ln -s /usr/lib/x86_64-linux-gnu/libstdc++.so.6 libstdc++.so.6
```

3.5 Setting paths

You need to set the directories to the MOOS binaries to your PATH environmental variable so that your operating system can find them. The procedure for this depends on your login shell (`> echo $SHELL`) if you're unsure.

bash (open `~/.bashrc`) or zsh (open `~/.zshrc`):

```
#path

export PATH=~/lamss/bin:\
~/lamss/scripts:\
~/lamss-shared/bin:\
~/lamss-internal/bin:\
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

```
export IVP_BEHAVIOR_DIRS=~/lamss/lib:\
~/lamss-shared/lib:\
~/lamss-internal/lib
```

tcsh: open (or create) `~/.cshrc` and enter the following:

```
set path = (~/lamss/bin)
set path = ($path ~/lamss/scripts)
set path = ($path ~/lamss-shared/bin)
set path = ($path ~/lamss-internal/bin)
set path = ($path /usr/local/sbin /usr/local/bin /usr/sbin /usr/bin /sbin /bin)

setenv IVP_BEHAVIOR_DIRS '~/lamss/lib'
setenv IVP_BEHAVIOR_DIRS '$IVP_BEHAVIOR_DIRS:~/lamss-shared/lib'
setenv IVP_BEHAVIOR_DIRS '$IVP_BEHAVIOR_DIRS:~/lamss-internal/lib'
```

Obviously, substitute your path for all cases of `~/...` above.

After doing all that, open a new terminal window (to refresh your .*rc file) and test the whole deal by typing the following command

```
> which MOOSDB pHelmIvP pAcommsHandler pCoroner iBellhop pNoiseSim
```

you should see something like:

```
/usr/bin/MOOSDB
/usr/bin/pHelmIvP
/usr/bin/pAcommsHandler
/home/toby/lamss/bin/pCoroner
/home/toby/lamss-shared/bin/iBellhop
/home/toby/lamss-internal/bin/pNoiseSim
```

Chapter 4

Running the LAMSS autonomy system

At this point, you have compiled the code and correctly configured your computer. This means you are ready to try running some simulation missions. The `missions-lamss` repository contains a set of scripts for building the MOOS-IvP configuration files and launching the backseat driver for LAMSS.

4.1 Quick Start

To get off the ground quickly, you can run a mission with a simulated Unicorn and Macrura (our two AUVs) and a topside by the following steps:

- check out the `missions-lamss` repository if you have not:

```
> cd ~/
> bzr co lp:lamss/missions-lamss
```

- start a simulated topside

```
> cd missions-lamss/topside
> ./simulation_launch.sh
```

- start a simulated Unicorn (you may wish to switch workspaces for each vehicle):

```
> cd missions-lamss/auv/unicorn
> ./simulation_launch.sh
```

- start a simulated Macrura

```
> cd missions-lamss/auv/macrura
> ./simulation_launch.sh
```

4.2 Configuration repository

The missions-lamss repository contains a complete set of MOOS ProcessConfig blocks for all relevant MOOS processes, each having extension .plug.

Similarly, there is a set of .plug files (bhv*.plug) for all IvP behaviors used in the LAMSS missions.

Key parameters (indicated by the bash-like symbols `$()`) in the process and behavior 'plugs' are replaced by environmental variables, defined in a set of definition files with extension .def.

At mission launch the plugs and definitions are combined into a fresh set of complete .moos and .bhv configuration files by a dedicated pre-processor `nsplug`.

The repository applies to both virtual experiments and field deployments, and as such provides maximal consistency between the two. Also, it ensures complete consistency in the communication infrastructure among vehicles and topside command and control. This is achieved by a hierarchial repository structure:

missions-lamss/global_plugs : Plugs common among all nodes, e.g communication.

missions-lamss/auv/auv_plugs . Plugs that are common to all AUVs.

missions-lamss/auv/macrura/macrura_plugs . Plugs that are specific to the AUV Macrura.

missions-lamss/cruise/current Cruise-specific plugs for current cruise, e.g local UTM datum, viewers

missions-lamss/topside/topside_plugs : Topside specific plugs, e.g pAcommsPoller, pTransponderAIS.

There is an additional level of granularity associated with the arrays and onboard processors on the vehicles. The current setup is for the MIT DURIP array, and the two nose mounted arrays SINGLE and DUAL.

The selection of MOOS-IvP processes and behaviors to be incorporated in the mission configuration is defined by `moos.meta`, also containing the mission launch using pAntler.

The behavior file is configured in a similar manner using `bhv.meta`.

4.3 Launching a mission

Cruise configuration

The first step is to configure the repository for a specific geographical area or cruise. The repository has inherited several cruise definitions from other programs, including glint08 and swamsi09. The configuration is performed by executing a script:

```
> cd ~/missions-lamss
> ./cruise_onfig.sh glint08
```

This script will link the symbolic subdirectory cruise/current to cruise/glint08 containing all the GLINT08-specific plugs and definitions, such as operations box, local UTM datum etc. It also should contains definitions of obstacles to be avoided by the vehicles, such as the research vessel and moorings.

The next step is to configure the command and report message set. For LAMSS there is a choice between the NaFCon message set used in PN07 or the new pAcommsHandler message set. A script is provided which chooses the associated plugs and definitions. For either message set, the prototype is set up for the MIT C2 topside, and using pAcommsHandler, currently the only fully operational configurations. To choose the fixed NaFCon message set, append "nafcon" to the desired launch script (for topside and all the AUVs):

```
> ./simulation_launch.sh nafcon
```

The pAcommsHandler LAMSS message set is the default so no parameters are needed:

```
> ./simulation_launch.sh
```

For the LAMSS default pAcommsHandler message set there are only minor differences in the functionality of the autonomy system. Both message sets contain Deploy and Prosecute commands, and Status, Contact, and Track reports. The commands differ only slightly, mainly in the flexibility of defining the geometry of the survey patterns provided by the LAMSS message set, as opposed to the fixed configuration of NaFCON, where loiter patterns, racetracks etc. have to be defined in the configuration files, i.e. the plugs decribed above.

The more significant difference is in the Contact Report, where the LAMSS message support the multitarget tracking.

Simulation with physical modems or emulators

For simulation of a single AUV and the topside, it may be advantageous to establish the modem connection through an actual modem pair or a modem emulator box.

In this case the simulation is launched using the flag `hw_modem` for both the virtual vehicle and the topside. This mode is also used when a simulated virtual vehicle is 'operating' in a real underwater network using an over-the-side modem, a very cost-effective approach to testing multivehicle collaborative autonomy.

A pool of 4 modems connected through an analog mixer has been established on `unicornsim.mit.edu` which can be accessed via tcp. The use of the modem pool is activated through use of the switch `tcp_modem`.

Acoustic array configuration

The configuration plugs for the acoustic arrays used for the AUV's are defined in the directory tree `/missions-lamss/auv/arrays/`. For example the configuration plugs for the MIT DURIP array are specified in the drirectory `/missions-lamss/auv/arrays/durip`, together with a definition file `array.def` containing array-specific parameters such as array spacing, frequency band, sampling rate, etc.

In each vehicle directory, a script is provided for associating a particular array with that vehicle. To select the DURIP array for the vehicle 'Unicorn', use the command sequence

```
> cd ~/missions-lamss/auv/unicorn
> ./array_config.sh durip
```

Starting a vehicle autonomy system

Go into the directory for the vehicle, e.g.

```
> cd ~/missions-lamss/auv/unicorn
```

For launching a LAMSS passive sensing simulation with high-fidelity array and acoustic simulation and onboard passive signal processing:

```
> ./simulation_launch.sh passive
```

To alternatively launch a simulation with the low-fidelity multi-target tracking simulator

```
> ./simulation_launch.sh bearingsim
```

For launching a simulation of a SWAMSI multi-static active acoustic sensing mission:

```
> ./simulation_launch.sh active
```

When using a physical modem or modem emulator, the same mission is started using the command

```
> ./simulation_launch.sh active hw_modem
```

Finally, to launch the actual vehicle autonomy system for field missions (on the vehicle):

```
> ./runtime_launch.sh
```

The same parameters (active, passive, nafcon) available for simulation can be appended to `runtime_launch.sh` as well.

To launch the vehicle-side node display, automatically configured for the mission launched:

```
> matlab -nojvm -nosplash < misc/small_uVis.m
```

launching the topside command and control

To start the topside with the iCommander Command console and the situational display:

```
> cd missions-lamss/topside
> ./simulation_launch.sh
```

To launch the Lat-Long to UTM converter tool, automatically configured for the current UTM grid and operations area:

```
> matlab

>> misc/geo_convert
```

Configuration Switches

The currently implemented global switches used with the `simulation_launch.sh` or `runtime_launch.sh` launch scripts are as follows

passive Activates the passive sonar acquisition system in the runtime mode and the towed array and passive acoustic simulators in the simulation mode, as well as the entire passive acoustic processing chain, including beamformer, bearing tracker etc.

Chapter 5

Writing a new MOOS process

5.1 Creating the code

MOOS processes are standalone applications that run an extension of `CMOOSApp`[1]. For details on how to write a MOOS process I refer you to Newman's document[2]. There are a number of excellent books out on C++ programming[3,4,5].

To save you the effort of creating the files from scratch, we have a shell script that generates a barebones MOOS process (does nothing but compile). In `lamss-internal/src/moos` run the shell script

```
> createMOOS_tes new_moos_process_name your_name your_email
```

There is a loose naming convention for MOOS processes: lowercase prefix (p = normal MOOS process, u = post mission utility, i = interface to non MOOS software / driver, uSim = simulation utility, etc.) followed by name (first letter capitalized). For example, let us create a new MOOS process that samples some oceanographic quantity called pSampler.

`createMOOS_tes pSampler` creates the following files:

```
pSampler:
     CMakeLists.txt
     pSampler.cpp
     pSampler.hn
     pSamplerMain.cpp
```

[1] http://www.robots.ox.ac.uk/~pnewman/MOOSDocumentation/Source/Core/MOOSLIB/html/classCMOOSApp.html
[2] http://www.robots.ox.ac.uk/%7Epnewman/MOOSDocumentation/ProgrammingWithMOOS/latex/ProgrammingWithMOOS.pdf
[3] http://www.amazon.com/Primer-Plus-5th-Stephen-Prata/dp/0672326973
[4] http://www.amazon.com/C-Standard-Library-Tutorial-Reference/dp/0201379260
[5] http://www.amazon.com/Cookbook-Cookbooks-OReilly-Ryan-Stephens/dp/0596007612

```
pSampler.moos
README
```

You will need to add the following line to the `lamss-internal/src/moos/CMakeLists.txt`:

```
add_subdirectory(pSampler)
```

To test the generation of the template, compile `lamss` as shown in section 3.4 (in `lamss/build` type `> cmake ..; make`).

Let me explain what these files contain:

- `CMakeLists.txt`: This tells the compiler what libraries to link to your code[6]. The basic MOOS libraries and some other commonly used ones are included here. Add to the list as you need them.
- `pSamplerMain.cpp`: Just declares the function main which instantiates CpSampler and calls CMOOSApp::Run on it. Typically you will not need to change anything here. Note that the convention for command line arguments (which are passed to argv within `int main`) should be

  ```
  > pSomeMoosProcess SomeMoosFile.moos SomeNameToRegisterToMOOSDB
  ```

- `pSampler.cpp`: Contains the actual implementation of the CpSampler class (which is derived from CMOOSApp). This is where the bulk of your code will go.
- `pSampler.h`: Contains the header declaration of the CpSampler class, which should include any member variables.
- `pSampler.moos`: An example .moos file for pSampler. You should fill this out with any and all configuration parameters your code accepts.
- `README`: A quick readme for lost souls who wander into your code directory.

5.2 Listing dependencies of your code

To ensure that another user will be able to build your code on their machine, you must make a note of any software packages that your code requires. This is done by updating the dependencies file (located at `lamss/DEPENDENCIES`), as follows:

If your dependency has the same package name for all Ubuntu distributions since the newest LTS (currently 10.04), list any commands that must be run to install dependencies under

[6] technically, it tells *CMake* how to make a Makefile to tell *make* how to tell the compiler (*g++*) what to link, but that's all a bit confusing.

"all". If the package name changed from the LTS to now, you must list your dependency correctly under "maverick", "lucid", "natty", etc. For most users, this will be similar to the following example, where `some-package` and `some-other-package` are the names of the Ubuntu packages to be installed.

```
all:
        @echo
        @echo "Installing yourname dependencies"
        @apt-get install \
                some-package \
                some-other-package \
                -y
#               we end with a "-y" so that each package line ends with a backslash
#               also, it makes sense that the user shouldn't get prompted
```

Note that the leading whitespace in this example is made of tabs, not spaces.

Save this file, and test your dependency-installer code by running:

```
> sudo ./dependencies.sh
```

Chapter 6

Documentation

A fair amount of documentation exists for MOOS outside of this starter paper:

- MOOS documentation: http://www.robots.ox.ac.uk/~pnewman/MOOSDocumentation/index.htm
- MOOS-IvP documentation: http://www.moosivp.org/docs.html **or** via SVN for a more up to date version:

  ```
  > svn co svn+ssh://{user}@oceanai.mit.edu/home/svn/repos/moos-ivp-doc moos-ivp-doc
  > cd moos-ivp-doc
  (maybe > sudo apt-get install transfig)
  > make pdf
  > gnome-open *.pdf
  ```

- MIT-MOOS documentation: via SVN:

  ```
  > svn co svn+ssh://{user}@oceanai.mit.edu/home/svn/repos/moos-ivp-doc moos-ivp-doc
  > cd moos-ivp-doc/memo_nested
  > make pdf
  > gnome-open *.pdf
  ```

  also investigate the documents in the other `/moos-ivp-doc/memo_*` folders.

- some stuff in `lamss/share/doc`

Chapter 7

Communication

Three listservers exist for MOOS @ LAMSS related questions, each with a different scope. They are listed below from largest to smallest scope.

- moosusers: https://lists.csail.mit.edu/mailman/listinfo/moosusers. use this for postings relevant to the entire MOOS community (not specific to MOOS-IvP or MIT).
- moosivp: https://lists.csail.mit.edu/mailman/listinfo/moosivp. use this for postings relevant to the MOOS-IvP community (not specific to MIT).
- moosivpmit: http://mailman.mit.edu/mailman/listinfo/moosivpmit. use this for postings specific to MIT moos users.
- lamss: http://mailman.mit.edu/mailman/listinfo/lamss. use this for postings specific to LAMSS (internal).

Chapter 8

Technical notes and things to watch out for

I have compiled a list of common errors and problems I (and others) have come across in the hopes of avoiding these in the future:

- When reading a MOOS file, you should use `m_MissionReader.GetValue` ONLY if you want to read a global parameter in the MOOS file. That is, if you want to read `LatOrigin`, `LongOrigin`, etc. In all other cases, you want to use `m_MissionReader.GetConfigurationParam` or `m_MissionReader.GetConfiguration`[1]. Consider the following MOOS file:

  ```
  LatOrigin = 10.23
  LongOrigin = 45.123
  ProcessConfig = pSampler
  {
      port = /dev/ttyS3
  }
  ProcessConfig = iGPS
  {
      port = /dev/ttyS4
  }
  ```

  Inside of iGPS, if you use `m_MissionReader.GetValue("port", gps_port)`, you will get the (probably) first instance of `port=` in the document put into `gps_port` (which is `/dev/ttyS3`) and thus NOT what you want. If instead you use `m_MissionReader.GetConfiguratonParam("port", gps_port)` you will get what you want (`/dev/ttyS4`).

[1] http://www.robots.ox.ac.uk/~pnewman/MOOSDocumentation/Source/Core/MOOSGenLib/html/classCProcessConfigReader.htm

- **Compiling problems**: If you have problems compiling your code, you probably have one of several things wrong:

  1. You do not have an external package you needed on your machine. See section 3.1 for details on all the packages you will need for MIT-MOOS. For example, if I did not have `libboost-dev` installed, I would see this error:

     ```
     ...
     moos-ivp-local/src/tes/pAcommsHandler/pAcommsHandler.cpp:
       In member function 'bool CpAcommsHandler::publish_incoming_data(std::string, int)':
     moos-ivp-local/src/tes/pAcommsHandler/pAcommsHandler.cpp:422:
       error: 'boost' has not been declared
     moos-ivp-local/src/tes/pAcommsHandler/pAcommsHandler.cpp:457:
       error: 'boost' has not been declared
     make[2]: *** [tes/pAcommsHandler/CMakeFiles/pAcommsHandler.dir/pAcommsHandler.o] Error 1
     make[1]: *** [tes/pAcommsHandler/CMakeFiles/pAcommsHandler.dir/all] Error 2
     make: *** [all] Error 2
     ```

     To remedy I simply install the missing package (`> sudo apt-get install libboost-dev` in this case)

  2. Someone has committed bad code. Email the author. Be sure to make use of `> make -k` if you need to build most of the project, ignoring the broken part.

Chapter 9

Migrating from the old moos-ivp-local build system

In April 2011, we moved from a subversion based system in the `moos-ivp-local` source tree to a bazaar based system hosted on launchpad.net. If you were a user of our code prior to this date, but have not yet updated, the following is a step by step guide for moving to this new system. If you are a new user, you can ignore this chapter.

9.1 Getting around Launchpad.net

Our project page is https://launchpad.net/lamss. You'll need to sign up for launchpad (https://launchpad.net/lamss/+login), and send someone on the `LAMSS Control` team[1] an email requesting permission to join the LAMSS team on launchpad. Please include your short launchpad name (e.g. mine is `tes`), which is listed under "Launchpad Id" at https://launchpad.net/people/+me.

Next you need to register your SSH public key with launchpad. If you already have a public key it's probably called `~/.ssh/id_rsa.pub`. If not, create one with `> ssh-keygen`. Now, copy the contents of the key (`id_rsa.pub`) to https://launchpad.net/people/+me/+editsshkeys.

That's all you have to do on launchpad.net right now, but if you're eager to click around you're welcome to. Also check out their user guide: https://help.launchpad.net/.

9.2 Setting up bazaar

If you don't have bazaar (`> which bzr`), install it with `> sudo apt-get install bzr`. Then, let `bzr` know who you are (use the same email you used on launchpad.net) and what your launchpad name is:

[1] https://launchpad.net/~lamss-control/+members#active

Old SVN Name (oceanai)	New bzr name (launchpad, aka lp)
svn+ssh://oceanai/.../moos-ivp-local/trunk/src	lp:lamss/lamss
svn+ssh://oceanai/.../moos-ivp-local/trunk/src-shared	lp:lamss/lamss-shared
svn+ssh://oceanai/.../moos-ivp-local/trunk/src-lamss	lp:lamss/lamss-internal
svn+ssh://oceanai/.../missions-lamss	lp:missions-lamss

Table 9.1: Mapping of old repositories to new

```
bzr whoami "Toby Schneider <tes.aubergine@gmail.com>"
bzr lp-login tes
```

9.3 Moving old moos-ivp-local / missions-lamss out of the way

Move the old moos-ivp-local / missions-lamss to another folder (or delete them)

```
mkdir old-lamss
mv moos-ivp-local missions-lamss old-lamss
```

9.4 Checkout new repositories

See table 9.4 for a mapping of the old repository sections to new repository names.

```
bzr co lp:lamss/lamss
bzr co lp:lamss/lamss-shared
bzr co lp:lamss/lamss-internal
bzr co lp:lamss/missions-lamss
```

9.5 Update dependencies

Please run:

```
cd lamss
sudo ./dependencies.sh
```

9.6 Build

Running `build.sh` from any of `lamss`, `lamss-shared`, `lamss-internal` builds all three in the proper order (`lamss-internal` depends on `lamss-shared` and `lamss`; `lamss-shared` depends on `lamss`).

Old Location	New Location
moos-ivp-local/src*/{user_name}/pMoosApp	lamss*/src/moos/pMoosApp
moos-ivp-local/src*/{user_name}/some_tool	lamss*/src/tools/some_tool
moos-ivp-local/src*/lib_my_lib	lamss*/src/lib/lib_my_lib
moos-ivp-local/src/matlab	lamss-shared/src/matlab
moos-ivp-local/scripts	lamss*/scripts
moos-ivp-local/docs/**/*.tex	lamss*/src/doc/*.tex
moos-ivp-local/docs/**/*.pdf	lamss*/share/doc/*.pdf (autogenerated)
moos-ivp-local/build (autogenerated)	lamss*/build (autogenerated)
moos-ivp-local/lib (autogenerated)	lamss*/lib (autogenerated)
moos-ivp-local/bin (autogenerated)	lamss*/bin (autogenerated)
moos-ivp-local/**/lib*/*.h	lamss*/include (autogenerated)
moos-ivp-local/missions	(deprecated, use missions-lamss)
moos-ivp-local/data	(deprecated, use missions-lamss)

Table 9.2: Mapping of moos-ivp-local folders to new lamss* folders. lamss* refers to lamss, lamss-internal or lamss-shared. See table 9.4. Otherwise * refers to a wildcard match of 0 or more, and ** refers to a recursive wildcard match.

9.7 Update Paths

See section 3.5 for the proper settings for your `PATH` (or `path`) environmental variable so that your system can find the `lamss` binaries.

9.8 Significant changes

`missions-lamss` is essentially unchanged. The SVN revision used for `missions-lamss` was 507. You will have to port changes newer than -r 507 yourself. These changes refer to `moos-ivp-local` and are summarized in table 9.8.

- I only ported the applications currently used by `missions-lamss/auv/unicorn`, `missions-lamss/auv/macru` `missions-lamss/auv/spermwhale` and `missions-lamss/topside`. Others you want to continue developing you should port over yourself.
- `lamss*` were created from `moos-ivp-local` revision 3258.
- There are no more user-name folders like `tes`, `henrik`, etc. Instead all `MOOS` apps reside in `lamss*/src/moos`
- Library source code folders have a true home now in `lamss*/src/lib`
- Document source code (that is, LaTeX) resides in `lamss*/src/doc` and is built to `lamss/share/doc`

- Include headers (`*.h`) from `lamss*/src/lib` are copied by `cmake` to `lamss*/include/lamss*` for use by others. As before, libraries are built to `lamss*/lib`, binaries are built to `lamss*/bin`. This means that `lamss*/` mimics `/usr/local` or `/usr`. `make install` copies `lamss*/share`, `lamss*/bin`, `lamss*/lib`, and `lamss*/include` to the corresponding `/usr/local` folder.

- MOOS Headers must be qualified from `MOOS/Core` or `MOOS/Essentials`. That is,

```
// yes!
#include "MOOS/libMOOS/MOOSLib.h"
#include "MOOS/libMOOS/Utils/MOOSUtils.h"
#include "ivp/MOOSGeodesy.h

// no!
#include "MOOSLib.h"
#include "MOOSGenLib.h"
#include "MOOSGeodesy.h
```

- Headers for `lamss*` (previously `moos-ivp-local`) libraries must be fully qualified from `lamss*/include`. That is,

```
// yes!
#include "lamss/lib_henrik_util/Measurement.h"
#include "lamss/lib_newmat10D/newmatap.h"
#include "lamss-shared/lib_bellhop/iBellhop_messages.pb.h"
#include "lamss-internal/lib_ais/OM1371Message.h"

// no!
#include "Measurement.h"
#include "newmatap.h"
#include "iBellhop_messages.pb.h"
#include "OM1371Message.h"
```

- Libraries must be linked (especially if they reside in non-standard locations like MOOS-IvP) using the CMake variable name, not the Linux .so name. That is,

```
// yes!
TARGET_LINK_LIBRARIES(pCoroner
  ${MOOS_LIBRARIES}
  ${IVP_MBUTIL_LIBRARY})

// no!
TARGET_LINK_LIBRARIES(pCoroner
```

```
  MOOS MOOSGen MOOSUtility
  mbutil)
```

You can run the following for a full list of available CMake variables:

```
cd lamss/build
cmake -LA .
```

For example, if I want all the IvP libraries, I can run

```
cd lamss/build
cmake -LA . | egrep "IVP.*LIBRARY"
```

which gives me

```
IVP_BEHAVIORS_LIBRARY:FILEPATH=/home/toby/moos-ivp/lib/libbehaviors.a
IVP_BHVUTIL_LIBRARY:FILEPATH=/home/toby/moos-ivp/lib/libbhvutil.a
IVP_CORE_LIBRARY:FILEPATH=/home/toby/moos-ivp/lib/libivpcore.a
IVP_GENUTIL_LIBRARY:FILEPATH=/home/toby/moos-ivp/lib/libgenutil.a
IVP_GEOMETRY_LIBRARY:FILEPATH=/home/toby/moos-ivp/lib/libgeometry.a
IVP_HELMIVP_LIBRARY:FILEPATH=/home/toby/moos-ivp/lib/libhelmivp.a
IVP_IPFVIEW_LIBRARY:FILEPATH=/home/toby/moos-ivp/lib/libipfview.a
IVP_IVPBUILD_LIBRARY:FILEPATH=/home/toby/moos-ivp/lib/libivpbuild.a
IVP_LOGIC_LIBRARY:FILEPATH=/home/toby/moos-ivp/lib/liblogic.a
IVP_LOGUTILS_LIBRARY:FILEPATH=/home/toby/moos-ivp/lib/liblogutils.a
IVP_MARINEVIEW_LIBRARY:FILEPATH=/home/toby/moos-ivp/lib/libmarineview.a
IVP_MBUTIL_LIBRARY:FILEPATH=/home/toby/moos-ivp/lib/libmbutil.a
IVP_NAVPLOT_LIBRARY:FILEPATH=/home/toby/moos-ivp/lib/libnavplot.a
```

If your third party library is not already listed, you'll want to add a `Find*.cmake` script for it in `lamss*/cmake_modules`.

Chapter 10

Bathymetry and CTD Data

10.1 CTD

MSEAS is our main source for CTD data. MSEAS has real data from sea experiments. In order to obtain it, write an email to someone in the MSEAS lab and ask for CTD data for specific month and Lat-Long coordinates. You will be given a .mods file. In order to read this file, run the modsread.m script.

modsread.m will do the following:

- Generate sound speed profile, temperature, and salinity vs depth
- Generate a plot from where these points were taken.
- Create .bin file

uSimCTD looks for the closest one of these points in 3D.

10.2 Bathymetry

The main source for bathymetry data is http://www.ngdc.noaa.gov/ngdc.html. In order to get the correct bathymetry data follow the following steps:

- Look in cruise.def and find the appropriate Latitude and Longitude for the area of interest
- Go to the NOAA stie and click "Bathy"
- Click on Bathymetric and Fishing maps
- Find appropriate Lat-Long corners and open PDF

In order to get a .xyz text file:

- Click on "Create Custom Grid"
- Click "x,y,z" with "no reader" and "no tab"
- If you want only sea information, click "advanced" and then click "sea cells"
- Name a Grid-ID and submit
- Save the .bin file in the cruise environmental folder