Launching Processes and Running Mission Scripts with `pAntler`

Paul Newman

August 13, 2009

Abstract

This document tells you how to use the application `pAntler` to launch multiple MOOS processes. This is a useful tool for starting up a whole bunch of processes, all of which share a single configuration file but which can be distributed over multiple machines and operating systems.

Contents

1 Synopsis

The process pAntler is used to launch/create a MOOS community. It is simple to use and post V7.0.2 very extensible.

One of the ideas underlying MOOS is the one mission file one mission paradigm. A single mission file contains all the information required to configure all the processes needed to undertake the task (mission) in hand [1]. Note a collection of MOOS processes is commonly referred to as a "community".

1.1 Basic Syntax

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

```
pAntler Mission.moos
```

will launch the required processes/community for the mission.

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

Run = *procname* [@ LaunchConfiguration] [∼ MOOSName]

where LaunchConfiguration is an optional comma-separated list of "parameter=value" pairs which collectively control how the process "procname" (for example iGPS, or iRemote or MOOSDB) is launched. Exactly what parameters can be specified is detailed later in the document.

Antler looks through its entire configuration block and launches one process for every line which begins with RUN= . When all processes have been launched, Antler waits for all of them to exit and then quits itself.

2 Controlling Process Launch

Immediately after the "@" symbol in a RUN directive the user can supply a list of "parameter=value" pairs (comma-separated), which control how the process in question should be launched. The following subsections will explain the action of available parameters.

2.0.1 Launching Processes in New Console Windows (or not)

```
Run = MOOSDB @NewConsole = true
```

The optional NewConsole parameter specifies whether the named process should be launched in a new window (an xterm in Unix or cmd-prompt in Win32 derived platforms). By default a new console is launched.

[1] And the pLogger application backs up each mission file so you know exactly what mission file was run at the time data was recorded — see pLogger documentation

2.1 Controlling Console Appearance

Post V7.0.2 releases allow a good deal of control over the appearance of the windows in which processes will be launched. This is especially so on the 'nix side of life [2]

By specifying `XConfig=`<*Name*> or `Win32Config=`<*Name*> (depending on OS) the user can have `Antler` apply customisations to the new console in which a process is launched. For example:

```
Run = MOOSDB @NewConsole = true,XConfig=DBXConsoleSettings,Win32Config=
    DBW32ConsoleSettings
```

will cause `Antler` to search through its configuration block to find a line which begins with `DBXConsoleSettings =` . or `DBW32ConsoleSettings =` – depending on OS). What is to the left of the equality determines the appearance of the new console and is a function of the host operating system.

2.1.1 Console Appearance in Unix like OSs

In Unix-derived operating systems the appearance string (referenced by "XConfig") is a comma-separated list of parameters that would be used to configure an xterm. So, to continue by way of the DBConsoleSettings example, if the DBConsoleSettings was specified (on its own line) as follows

```
DBXConsoleSettings = -bg, \#FF0000,-fg,\#FFFFFF,-geometry,80x12+2+00,+sb,-
    T,TheMOOSDB
```

then the MOOSDB would be launched in 12 rows by 80 columns window, white text on red at the top left of the screen. The string "TheMOOSDB" would appear in the title. Note any xterm configuration parameters can be specified in this way. See the manual page for xterm for information on the options allowed.

2.1.2 Console Appearance in Win32 OSs

The only native Win32 console options supported control the background color of the terminal (text is always white). The LHS of the configuration line (referenced by "Win32Config") can contain a comma-separated list of `BACKGROUND_RED` `BACKGROUND_BLUE` and `BACKGROUND_GREEN` . In this way

```
DBW32ConsoleSettings = BACKGROUND_RED
```

would produce a white on red Win32 console.

2.1.3 Appearance Example

```
ProcessConfig = Antler
{
        //look on system path
        ExecutablePath = system
```

[2] The native Win32 console has less flexibility than the xterm. Deep apologies for Win32 users who may feel hard done by by the asymmetry here.

```
        //launch a DB
        Run = MOOSDB  @NewConsole = true,XConfig=DBXConsoleSettings,
            Win32Config=DBW32ConsoleSettings

          //xterm configuration for DB
          DBXConsoleSettings = -bg, \#FF0000,-fg,\#FFFFFF,-geometry,80x12
              +2+00,+sb,-T,TheMOOSDB

          //Win32 Configuration for DB
          DBW32ConsoleSettings = BACKGROUND_RED
}
```

2.2 Controlling Search Paths

Post V7.0.2 `Antler` offers extended functionality regarding specifying how executables are located on the host file system. The paths which you wish the OS to use when searching for the executable to launch can be specified globally (a common path for all processes) or on a process by process basis.

2.2.1 Specifying Global Executable Paths

Adding a line of the form

`ExecutablePath =` *path*

to Antler's configuration block where *path* is a suitable path string, will make `Antler` search in that place for the executables to launch. Not specifying this variable or setting path to "SYSTEM" will cause Antler to rely on the host OS being able to locate the executable in its own executable paths.

2.2.2 Specifying Paths for an Individual Process

The global executable path (default "system") can be overridden for a particular process by providing your preferred path in the "RUN" directive line. For example

```
Run = pP1          @  NewConsole = true,path=/usr/strangeplace
```

will try to launch a process called "pP1" from a the directory "/usr/strangeplace". Such process-specific path directives override any path set with `ExecutablePath=...` (See Section 2.2.1).

3 Passing Parameters to Launched Processes

3.1 The Two Default Parameters

Unless told otherwise (see Section 4.1.2) each process launched is passed the mission file name as a command line argument and also the name it should use to register with the MOOSDB. This means that by default `argv[1]` of `main`

is the name of the mission file currently in play (the one which `pAntler` is itself reading) and argv[2] is the name of the process to be launched (for example `iGPS` or `pLogger`). By default pAntler assumes the name with which a process will be registering with the MOOSDB is the name of the process itself. For example, `pLogger` will register with the MOOSDB with the name "pLogger". However this can be changed using the ~ `MOOSName` syntax:

```
Run = iGPS @NewConsole = true ~ GPS_A
```

will cause the executable called "iGPS" to be launched in a new console but (because iGPS handles command line parameters appropriately) it will register with the MOOSDB under the name of "GPS_A".

3.1.1 Handling Default Parameters

Of course just passing the MOOSName to a process doesn't mean automatically that all MOOS connections within that process will use this name. Supporting code must be provided. Listing 1 illustrates just one way in which this can be done.

Listing 1: Handling default command line parameters. Note how the MOOSName and Mission file are passed to the CMOOSApp derived object.

```
#include "SimpleApp.h"

//simple "main" file which serves to build and run a CMOOSApp-derived
    application


int main(int argc ,char * argv[])
{
    //set up some default application parameters

    //whats the name of the configuration file that the application
    //should look in if it needs to read parameters?
    const char * sMissionFile = "Mission.moos";

    //under what name should the application register with the MOOSDB?
    const char * sMOOSName = "MyMOOSApp";

    switch(argc)
    {
    case 3:
        //command line says don't register with default name
        sMOOSName = argv[2];
    case 2:
        //command line says don't use default "mission.moos" config file
        sMissionFile = argv[1];
    }

    //make an application
    CSimpleApp TheApp;
```

```
    //run forever passing registration name and mission file parameters
    TheApp.Run(sMOOSName, sMissionFile);

    //probably will never get here..
    return 0;
}
```

4 Running Multiple Instances of a Particular Process

As already described in Section 3, the optional `MOOSName` parameter allows MOOSProcesses to connect to the `MOOSDB` under a specified name. Why is this useful? Well, for example, a vehicle may have two GPS instruments onboard. Now by default `iGPS` may register its existence with the `MOOSDB` under the name "iGPS". This name is now taken and no other MOOSClient can use the name "iGPS"[3]. By using the ~ syntax multiple instances of the executable `iGPS` can be run but with each connecting to the `MOOSDB` using a different name. For example

```
Run = iGPS @ NewConsole = true ~iGPSA
Run = iGPS @ NewConsole = true ~iGPSB
```

would launch two instances of `iGPS` registering under "iGPSA" and "iGPSB" respectively. **Note there would need to be *two* GPS configuration blocks in the mission file – one for each – and the process names (RHS of `ProcessConfig=`) would be "iGPSA" and "iGPSB".**

4.1 Customising the Command Line Parameters Passed to a Launched Process

But what if your beloved new process which you desire `Antler` to launch requires extra command line configuration? Or what if you don't want `Antler` to pass the Mission file name and the MOOS name as run time (command line) parameters? Fear not, just read on.

4.1.1 Specifying Additional Process Command Line Parameters

You can specify additional parameters which should be passed to a launched process using a syntax similar to that used to specify console appearance (see Section 2.1). The trick is to specify the name of a parameter string (RHS of a `ExtraProcessParams=...` in the process's `RUN` directive line. `Antler` then rescans its configuration block looking for this named string, which must be a comma-separated list of parameters. An example will make this blindingly obvious.

```
ProcessConfig = Antler
{
        Run = iProcA @ NewConsole = true,path=/usr/local/bin,
            ExtraProcessParams=ProcAParams
```

[3]If they try, the MOOSDB will not accept them into the fold.

```
        ProcAParams =-o,--verbose,--clever
}
```

The above would launch a process called ‘ ‘iProcA” in a new console, (with default appearance as no appearance string is specified, see Section 2.1), and the process will be passed **six** parameters at launch time:

argv[0] the executable image name

argv[1] the mission file name

argv[2] the process's MOOS name

argv[3] -o

argv[4] –verbose

argv[5] –clever

4.1.2 Inhibiting Default Parameters and Launching Arbitrary (non-MOOS) Processes

If you want to launch a process with `Antler` that has not been designed to handle the mission file name and MOOS name as the first two parameters passed in the command line, then it is possible to tell `Antler` not to pass these parameters. This is done using the `InhibitMOOSParams` key word. For example, if you wanted to launch the executable `top` in its own window, you would use configuration similar to that in Listing 2.

Listing 2: Launching a non-MOOS process-like top (here on a 'nix system). Note the use of `InhibitMOOSParams`

```
ProcessConfig = Antler
{
Run = top @ NewConsole = true,path=system, InhibitMOOSParams=true, XConfig
    = TopX, ExtraProcessParams=TopParams

 //xterm configuration
 TopX = -bg, #F00000,-fg,#FFFFFF,-geometry,80x40+200+300,+sb,-T,top_top

  //list of parameters
  TopParams =-o,cpu
}
```

.

4.2 Controlling Verbosity

By default `pAntler` gives hearty feedback about what it is launching, this feedback cane be limited or completely removed by launching `pAntler` with the `--verbose` , `--terse` or `---quiet` flags on the command line. For example

```
./pAntler GoToTheMOOS.moos --terse
```

5 Distributing a Community over Multiple Machines

5.1 Motivation

Up until now we have implicitly assumed that all processes launched by a single instance of `Antler` reside on the same physical computer. Surely this conflicts with the idea that any MOOS process can run on any machine under any (common) OS? You're right, it does, and this issue has been addressed in post V7.0.2 versions. Excellent. In the broadest of terms, it is possible to have one Antler send a single mission file to a host of other Antlers (presumably, but not necessarily, sitting on a different machine or OS), which they then process and launch processes locally. The idea is that you still only need to edit one mission file to control a suite of processes running over any number of physical machines. The operating paradigm is that once a suitably configured `Antler` has been started on a machine, you need never kill or restart it. It stays alive patiently waiting for instructions. See Figure 5.1.

5.2 Antler Modes: Monarch and Headless

The idea is that `Antler` can be run in one of two modes, which we shall refer to as "headless" and "monarch"[4]. These terms only have meaning if the `EnableDistributed` flag is set to `true` in the Antler configuration block – i.e. when Antler is being told to support distributed process control. If this flag is set and `Antler` is launched in the usual way:

```
./pAntler Mission.moos
```

then this will become a "Monarch". Think of it as the king/governing/-top/controlling `Antler` which will take responsibility for distributing (via the MOOSDB) the mission file to any other "headless" Antlers sitting on other machines. If however you start `Antler` with three command line parameters as follows:

```
./pAntler lisa1.robots.ox.ac.uk 9000 lisa2
```

then Antler will launch in "headless" mode. Headless `Antlers` are bound to a single "Monarch" via a MOOSDB (which will usually be launched by the monarch itself.) The three parameters specify the location and port of this MOOSDB and also an AntlerID. This last parameter is a string which is used by headless Antlers to figure out which Run directives they should execute.

Consider the following simple example:

```
ProcessConfig = Antler
{
        EnableDistributed = true
        Run = iProcA @ AntlerID = lisa2, NewConsole = true
        Run = MOOSDB @ NewConsole=true
}
```

[4] As in Monarch of the Glen – referring to the size of antlers.

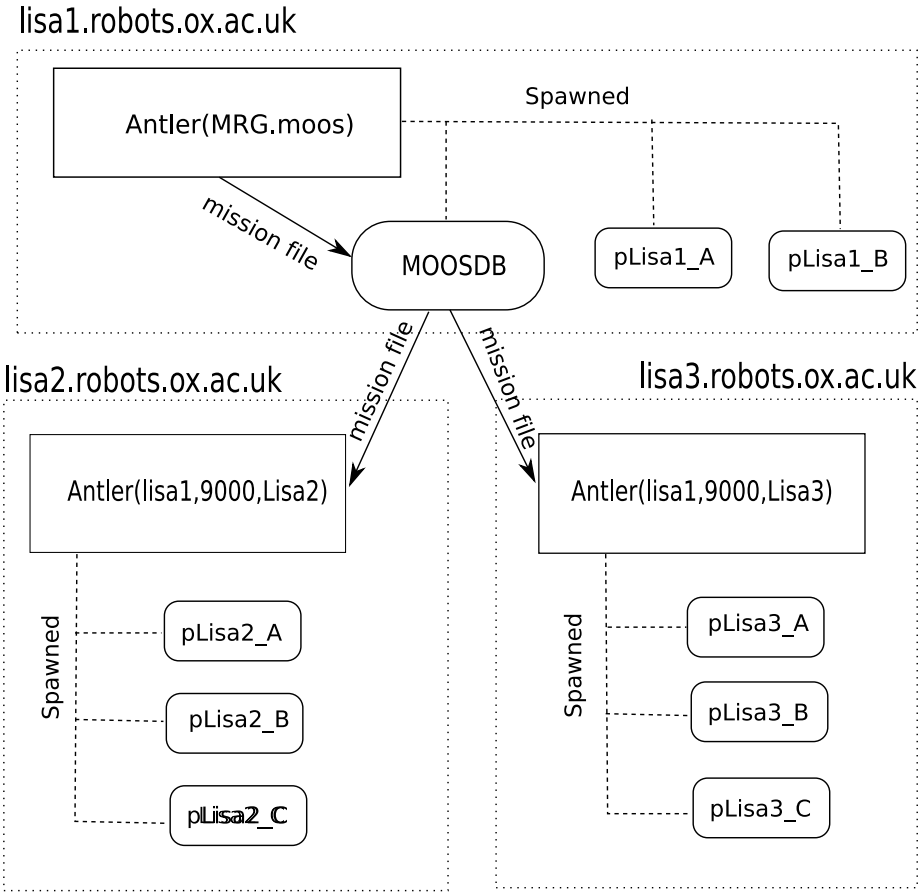

Figure 1: In distributed mode, `Antler` can be started in one of two ways. Here on the machine lisa1.robots.ox.ac.uk it is started in "Top MOOS" mode with the name of a mission file on the command line. On the two other machines (lisa2 and lisa3) Antler is started in headless mode receiving three command line parameters – the machine name on which a `MOOSDB` can be found, the port that `MOOSDB` is serving on and an AntlerID name, which in this case is simply set to the machine name. When the "topMOOS" has spawned its processes it pushes the mission file to the DB. The headless Antlers pick up this notification and run themselves from the newly received mission file. Each headless Antler only launches processes which have a Run directive line containing that particular instantiation of Antler's ID.

Note how `iProcA` has an `AntlerID` specified. Now if, as above, I started a headless `Antler` with "lisa2" as its Antler ID on machine "B" and then started another instance of `Antler` on machine "B" [5] but this time only specifying a mission file (i.e. start Antler as a "monarch") you would witness a MOOSDB coming up on machine A and iProcA starting on machine B. If no AntlerID is specified in a run directive, it is assumed that the monarch is required to process the directive. Headless `Antlers` only process run directives possessing an AntlerID matching their own. Each headless Antler writes the runtime received mission file (stripped of comments and superfluous white space) to local disk (working directory) under the name `dynamic_<TIMESTAMP>.moos` for future perusal.

5.2.1 Shutdown Behaviour

The default behaviour is for headless Antlers to shut down all their spawned processes when contact is lost with the MOOSDB. If this is not the desired behaviour and you want launched processes to carry on running, simply add the directive "KillOnDBDisconnect=false" to the configuration block.

```
ProcessConfig = Antler
{
        EnableDistributed = true
        KillOnDBDisconnect = false
        Run = iProcA @ AntlerID = lisa2 , NewConsole = true
        Run = MOOSDB @ NewConsole = true
}
```

In any case as soon as a mission file is received by a headless Antler any and all running processes will be shut down before processing the new mission file.

6 Examples

If you enable the building of examples via the CMake build screen (see Figure 6) then the example configurations in Sections 6.1 and 6.2 serve as a good starting point in experimenting with `Antler` . There are three example processes supplied in the sibling code directory of the documentation:

pAntlerTestAppA is nothing more than a dumb CMOOSApp that prints a string declared in its configuration block.

pAntlerTestAppB is nothing more than a dumb CMOOSApp which takes more than the standard two command line arguments; it uses these additional params to publish a variable to a MOOSDB.

pAntlerTestAppC is not a CMOOSApp. It is just a program which prints out its command line arguments and spins in a do nothing loop.

[5] A can equal B, but what is the point?

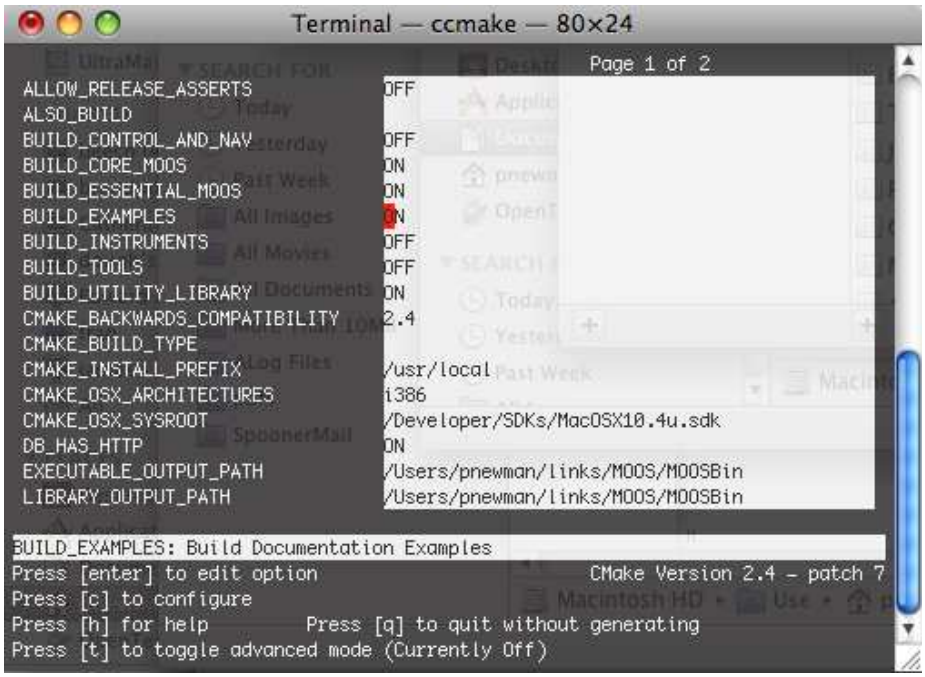

Figure 2: Selecting the building of examples in the MOOS build screen.

6.1 Local Configurations

Listing 3: Example Configuration Blocks for Antler, where all process are run on the same host machine. This file can be found in the sibling code directory of Antler in the documentation tree.

```
//Un-Comment/Comment the first line of each example block //to
play with various Antler configurations


//simplest possible example
//ProcessConfig = Antler
{
    Run = MOOSDB @ NewConsole = true
    Run = pAntlerTestAppA @ NewConsole = true

}

//Run two instances of pAntlerTestAppA under different names
//note two new configuration blocks are needed (Oxford and FenTech)
//ProcessConfig = Antler
{
    Run = MOOSDB @ NewConsole = true
    Run = pAntlerTestAppA @ NewConsole = true  ~ Oxford
    Run = pAntlerTestAppA @ NewConsole = true ~ FenTech
}
```

```
//passing an additional two parameters to pTestAppB
//ProcessConfig = Antler
{
    Run = MOOSDB @ NewConsole = true
    Run = pAntlerTestAppA @ NewConsole = true

    Run = pAntlerTestAppB @ ExtraProcessParams = BParams , NewConsole =
        true
    BParams = CustomVar, ThisIsAString

}

//specifying a default executable path and overloading it for MOOSBD
ProcessConfig = Antler
{
    ExecutablePath  = C:/codescratch/MOOS/MOOSBin/debug/q
    Run = MOOSDB @ path=C:/codescratch/MOOS/MOOSBin/debug, NewConsole =
        true
    Run = qq @ NewConsole = true

}


//passing three parameters to pTestAppC which is not expecting the first
//two parameters to be Mission File and MOOSName
//ProcessConfig = Antler
{
    Run = MOOSDB @ NewConsole = true

    Run = pAntlerTestAppA @ NewConsole = true

    Run = pAntlerTestAppB @ ExtraProcessParams = BParams , NewConsole =
        true
    BParams = CustomVar, ThisIsAString

    Run = pAntlerTestAppC @ ExtraProcessParams = CParams,
        InhibitMOOSParams=true , NewConsole = true
    CParams = set, the, moos, loose, 1, 2, 3, 45.6

}

//Adding some colour to MOOSDB, pAntlerTestB and pAntlerTestC
ProcessConfig = Antler
{
    Run = MOOSDB @Win32Config=DBWin32, XConfig=DBX , NewConsole = true
    DBX = -bg,#FF0000, -geometry, 80x40+200+300
    DBWin32 = BACKGROUND_RED

    Run = pAntlerTestAppA @ NewConsole = true

    Run = pAntlerTestAppB @ Win32Config=BWin32, XConfig=BX,
        ExtraProcessParams = BParams , NewConsole = true
```

```
    BParams = CustomVar, ThisIsAString
    BWin32 = BACKGROUND_GREEN, BACKGROUND_BLUE
    BX = -bg,#00FFFF, -geometry, 80x40+350+300

    Run = pAntlerTestAppC @ Win32Config=CWin32, XConfig=CX,
        ExtraProcessParams = CParams, InhibitMOOSParams=true , NewConsole
        = true
    CParams = set, the, moos, loose, 1, 2, 3, 45.6
    CWin32 = BACKGROUND_RED, BACKGROUND_BLUE
    CX = -bg,#FF00FF, -geometry, 80x40+400+300

}


//Configuration for TestAppA - just looks for a string to print
ProcessConfig = pAntlerTestAppA
{
    PrintThis = SetTheMOOSLoose
}

//configuration for pTestAppB - nothing but it expects a third
and fourth //command line to tell it what to publish...
ProcessConfig = pAntlerTestAppB {

}

//Configuration for FenTech (which is actually an
instantiation of // pAntlerTestAppA) - just looks for a string
to print ProcessConfig = FenTech {
    PrintThis = ThisIsTestAppAAsFenTech
}

//Configuration for Oxford (which is actually an instantiation of
// pAntlerTestAppA) - just looks for a string to print
ProcessConfig = Oxford
{
    PrintThis = ThisIsTestAppAAsOxford
}
```

6.2 Distributed Configuration

Listing 4: Example Configuration Blocks for Antler, where processes are run on different hosts. This file can be found in the sibling code directory of Antler in the documentation tree.

```
//if you are really runing this on different hosts
//make sure you set the server hostname below
//if you are simply testing how to run multiple
//instances of MOOS using localhost is just fine


ServerHost = host1.robots.ox.ac.uk
ServerPort = 9000
```

```
//you need to start three instances of pAntler...
//On host3: pAntler host1.robots.ox.ac.uk 9000 jupiter
//On host2: pAntler host1.robots.ox.ac.uk 9000 neptune
//On host 1: pAntler <MissionFile.moos>

//Running AppA on jupiter(host3) B on neptune(host2), DB and C on host1
ProcessConfig = Antler
{
    EnableDistributed = true


    Run = MOOSDB @Win32Config=DBWin32,XConfig=DBX , NewConsole = true
    DBX = -bg,#FF0000, -geometry, 80x40+200+300
    DBWin32 = BACKGROUND_RED

    Run = pAntlerTestAppA @ AntlerID = jupiter, NewConsole = true

    Run = pAntlerTestAppB @ AntlerID = neptune,Win32Config=BWin32,XConfig=
        BX, ExtraProcessParams = BParams , NewConsole = true
    BParams = CustomVar,ThisIsAString
    BWin32 = BACKGROUND_GREEN,BACKGROUND_BLUE
    BX = -bg,#00FFFF, -geometry, 80x40+350+300

    Run = pAntlerTestAppC @ Win32Config=CWin32,XConfig=CX,
        ExtraProcessParams = CParams, InhibitMOOSParams=true , NewConsole
        = true
    CParams = set,the,moos,loose,1,2,3,45.6
    CWin32 = BACKGROUND_RED,BACKGROUND_BLUE
    CX = -bg,#FF00FF, -geometry, 80x40+400+300

}

//As above but now with three versions of AppA
//ProcessConfig = Antler
{

    EnableDistributed = true

    Run = MOOSDB @Win32Config=DBWin32,XConfig=DBX , NewConsole = true
    DBX = -bg,#FF0000, -geometry, 80x40+200+300
    DBWin32 = BACKGROUND_RED

    Run = pAntlerTestAppA @ AntlerID = jupiter, NewConsole = true
    Run = pAntlerTestAppA @ AntlerID = jupiter, NewConsole = true ~
        FenTech
    Run = pAntlerTestAppA @ AntlerID = jupiter, NewConsole = true ~ Oxford

    Run = pAntlerTestAppB @ AntlerID = neptune,Win32Config=BWin32,XConfig=
        BX, ExtraProcessParams = BParams , NewConsole = true
    BParams = CustomVar,ThisIsAString
    BWin32 = BACKGROUND_GREEN,BACKGROUND_BLUE
    BX = -bg,#00FFFF, -geometry, 80x40+350+300
```

```
    Run = pAntlerTestAppC @ Win32Config=CWin32,XConfig=CX,
        ExtraProcessParams = CParams, InhibitMOOSParams=true , NewConsole
        = true
    CParams = set,the,moos,loose,1,2,3,45.6
    CWin32 = BACKGROUND_RED,BACKGROUND_BLUE
    CX = -bg,#FF00FF, -geometry, 80x40+400+300

}



//Configuration for TestAppA - just looks for a string to print
ProcessConfig = pAntlerTestAppA
{
    PrintThis = SetTheMOOSLoose
}

//configuration for pTestAppB  - nothing but it expects a third and fourth
//command line to tell it what to publish...
ProcessConfig = pAntlerTestAppB
{

}

//Configuration for FenTech (which is actually  an instantiation of
// pAntlerTestAppA) - just looks for a string to print
ProcessConfig = FenTech
{
    PrintThis = ThisIsTestAppAAsFenTech
}

//Configuration for Oxford (which is actually  an instantiation of
// pAntlerTestAppA) - just looks for a string to print
ProcessConfig = Oxford
{
    PrintThis = ThisIsTestAppAAsOxford
}
```

7 Application note : I/O Redirection – Deployment

Frequently `iRemote` , displayed on a remote machine, will be the only interface a user has to the MOOS community. We must ask the question – "where does all the I/O 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. [6]

Running `Antler` in the following fashion followed by a manual start up of `iRemote` is the recommended way of running MOOS in the field on a 'nix platform.

[6] Some OSs are good for development, others for running...

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).