Introduction to Programming with MOOS

Paul Newman

June 4, 2009

Abstract

This document is intended to bootstrap the process of using the MOOS communications libraries. It is aimed at programmers who are competent in C++. All the code in this document is supplied in full in the "code" sibling directory of this document.

Contents

1 Building a New Application using MOOS

We shall get straight to the matter in hand — how to use the MOOS/Core directory to build a new application which takes advantage of the MOOS communications layer. To do this, it makes sense to quickly introduce an important class provided by the MOOS Communications libraries, namely `CMOOSApp` .

1.1 Derivation from `CMOOSApp`

MOOS provides a class called *CMOOSApp* which makes writing a program using MOOS a simple affair. Beneath the hood of the `CMOOSApp` class is a loop which repetitively calls a function called `Iterate()` which by default does nothing. One of our jobs as writers of a new MOOS-enabled application is to flesh this function out with the code that makes the application do what we want[1]. Behind the scenes this uber-loop in `CMOOSApp` [2] is also checking to see if new data has been delivered to the application. If it has, another virtual function, `OnNewMail` , is called — this is the spot to write code to process the newly delivered data.

We'll cover this in more detail later, but for now it makes sense to get stuck into an example. But before we do, look at Figure 1 which summarises graphically the basic flow of execution in an application which makes use of the `CMOOSApp` class (by deriving a new class from it) and has just called the `CMOOSApp::Run` method.

2 A First Worked Example

So let us use `CMOOSApp` to build an new application. Perhaps the simplest procedure is as follows:

1. Make a new "main.cpp".
2. Make a new class derived class from *CMOOSApp*.
3. In main() make an instance of this class.
4. Call ::Run() on this object.
5. As needs dictate overload the following virtual functions:

   `::Iterate()` a function in which the application will do its main processing; see Section 3.1.

   `::OnNewMail()` a function called when new mail (data) has arrived; see Section 3.2

   `::OnConnectToServer()` called when a connection has been made to the MOOS database; see Section 3.3

   `::OnStartup()` called when the application starts up; see Section 3.5

[1] Don't write a forever loop in `Iterate()` - allow CMOOSApp to call this function for you time and time again.

[2] You can find the loop in the `CMOOSApp::Run` method.

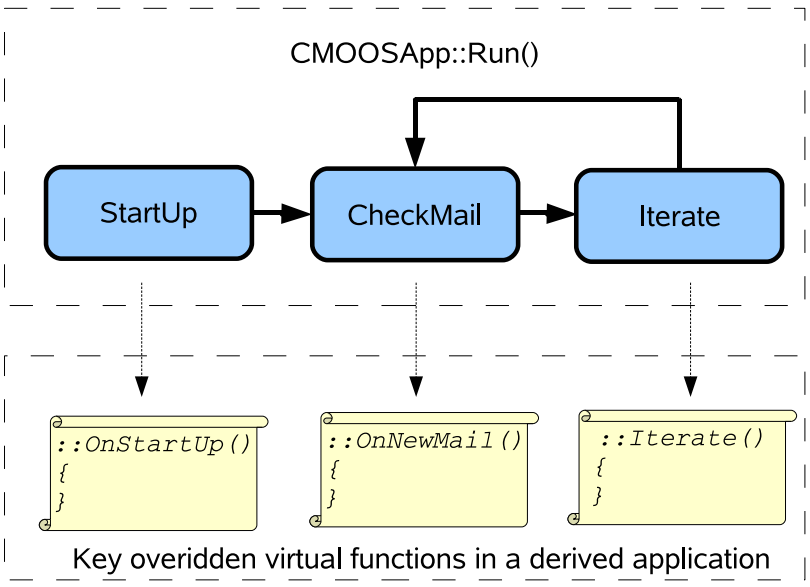

Figure 1: The flow of execution once `::Run` has been called on a class derived from `CMOOSApp` . The "scrolls" indicate where we (as users of the functionality of `CMOOSApp`) will be writing new code that implements whatever it is we want our application (program) to do.

Listings 1, 2 and 3 show the most skinny of conceivable MOOS applications using the functionality of `CMOOSApp` as a base class to a newly derived class `CSimpleApp` . My advice to you is to not proceed beyond this point until you can get this code to compile and link. See the document on Building and Linking for more details on building MOOS projects.

2.1 First Example Code Listing

Listing 1: Simplest Application - main.cpp

```
#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 shoud 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 pasing registration name and mission file parameters
    TheApp.Run(sMOOSName, sMissionFile);

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

Listing 2: Simplest Application - main.cpp

```
// Ex1/SimpleApp.h: interface for the CSimpleApp class.

#ifndef SIMPLEAPPH
#define SIMPLEAPPH

#include <MOOSLIB/MOOSApp.h>

class CSimpleApp : public CMOOSApp
{
public:
    //standard construction and destruction
    CSimpleApp();
    virtual ~CSimpleApp();

protected:
    //where we handle new mail
    bool OnNewMail(MOOSMSG_LIST &NewMail);
    //where we do the work
    bool Iterate();
    //called when we connect to the server
    bool OnConnectToServer();
    //called when we are starting up..
    bool OnStartUp();

};

#endif
```

Listing 3: Simplest Application - main.cpp

```
#include "SimpleApp.h"

//default constructor
CSimpleApp::CSimpleApp()
{
}
//default (virtual) destructor
CSimpleApp::~CSimpleApp()
{
}

/**
Called by base class whenever new mail has arrived.
Place your code for handling mail (notifications that something
has changed in the MOOSDB in this function

Parameters:
    NewMail :  std::list<CMOOSMsg> reference

Return values:
    return true if everything went OK
    return false if there was problem
**/
bool CSimpleApp::OnNewMail(MOOSMSG_LIST &NewMail)
{
    return true;
}

/**
called by the base class when the application has made contact with
the MOOSDB and a channel has been opened. Place code to specify what
notifications you want to receive here.
**/
bool CSimpleApp::OnConnectToServer()
{
    return true;
}

/** Called by the base class periodically. This is where you place code
which does the work of the application **/
bool CSimpleApp::Iterate()
{
    return true;
}

/** called by the base class before the first ::Iterate is called. Place
startup code here - especially code which reads configuration data from the
mission file **/
bool CSimpleApp::OnStartUp()
{
    return true;
}
```

3 The Important CMOOSApp Virtual Functions

`CMOOSApp` itself contains a few important virtual functions which can (should) be overridden in derived classes. These functions are called by the base class at the suitable time.

3.1 Iterate

By overriding the `CMOOSApp::Iterate` function in a new derived class, the author creates a function from which he or she can orchestrate the work that the application is tasked with doing. As an example, and without prejudice, imagine the new application was designed to control a marine vehicle. The iterate function is automatically called by the base class periodically and so it makes sense to execute one cycle of the controller code from this "`Iterate`" function. Some things to note here:

- Don't enter into an infinite loop waiting on data in this code - it won't break anything (the thread that handles the communications with other processes will still be running and responsive to you posting or checking for mail) but it is rather orthogonal to the intended use of `CMOOSApp`.
- You can configure the rate at which `Iterate` is called by the `SetAppFreq()` method or by specifying the "AppTick" parameter in a mission file (see Section 6.1 for more on configuring an application from a file).
- Note that the parameter passed to `SetAppFreq()` specifies the maximum frequency at which `Iterate` will be called - it does not guarantee that it will be called then - for example if you write code in iterate that takes 1s to complete there is no way that iterate can be called at more than 1Hz.
- If you want to call iterate as fast as is possible simply call `SetAppFreq(0)` — but ask yourself why you need such a greedy application, are you being polite?
- Although MOOSApp doesn't enter into a contract with you about exactly when `Iterate` will be called, it does allow you to know when it is being called. The function `MOOSTime` returns unix time in floating point seconds.

3.2 OnNewMail

This function is called from within `CMOOSApp::Run()` if and when some other process has posted data that you ("you" being the application here) have previously declared an interest in (see Section 4.1). The mail arrives in the form of a `std::list<CMOOSMsg>` — a list of `CMOOSMsg` objects (see Section 5). The programmer is free to iterate over this collection examining who sent the data, what it pertains to, how old it is, whether or not it is string or numerical data and to act / process the data accordingly.

3.3 OnConnectToServer

Unlike `Iterate` and `OnNewMail` this function is not called directly from `CMOOSApp::Run()`. It is actually a callback from a thread in the `m_Comms` object (a instance `CMOOSCommsObject`) possessed by `CMOOSApp` that handles all the IPC communications [3]. The callback occurs whenever contact has been made with the `MOOSDB` server which sits at the heart of the MOOS topology (see Section on "Topology" in the CommsArchitecture document.) This is one of two

[3] Indeed you could think of `CMOOSApp` as a fancy wrapper for this object.

places where the programmer is advised to call `m_Comms.Register` to tell the `MOOSDB` that we want to be sent mail if any other process posts data relating to a particular variable. See the example code and the sibling "CommsArchitecture" document and this will become blindingly obvious. Just remember that code executed in this callback is not in thread 0.

3.4 OnDisconnectFromServer

This is the counter part of `OnConnectToServer` . It is called when contact has been lost with the `MOOSDB` - generally, if this happens something terrible has happened. It is here for completeness. If there is nothing special you want to do when comms has been lost, don't bother adding this function to your `CMOOSApp` -derived class.

3.5 OnStartUp

This function is called by `CMOOSApp::Run` just before it enters into its own "forever-loop". This is the spot that you would populate with initialisation code, and in particular use the functionality provided by the `m_MissionReader` member object to read configuration parameters (including those that modify the default behaviour of the `CMOOSApp` base class) from file. See Section 6.1 for more details on configuration from file.

4 A Second Worked Example - Handling Mail

At present our application really doesn't do much - it just connects to the DB and sits there (behind the scenes the application is in regular contact with the DB but you wouldn't know it at the moment). So we'll now modify the Example 1 code and fill in the functions described in Section 3. The code in Main.cpp remains unchanged [4] so we won't replicate it here, but Listings 4 and 5 show the updated code for our own `MOOSApp` .

We shall pretend we are building some process running on a vehicle[5] — perhaps some kind of navigation process. Anyway, assume that this process needs to know about the status of the vehicle and its heading. We assume that some other processes (written by someone who has already read this document...) are publishing this data via the MOOS infrastructure. The question is, how do we get hold of this data? The first thing to do is "register" for mail and then write code in the `OnNewMail` function to parse the mail. We'll cover these topics in Sections 4.1 and 4.2.

Listing 4: Simplest Application - main.cpp

```
// Ex2/SimpleApp.h: interface for the CSimpleApp class.

#ifndef SIMPLEAPPH
#define SIMPLEAPPH

#include <MOOSLIB/MOOSApp.h>
```

[4] Apart from the fact that the MOOSApp is told to register as "Ex2" rather than "Ex1".

[5] That's just because I work on mobile robotics - there is nothing about the MOOS Comms API that is specific to autonomous vehicles.

```
class CSimpleApp : public CMOOSApp
{
public:
    //standard construction and destruction
    CSimpleApp();
    virtual ~CSimpleApp();

protected:
    //where we handle new mail
    bool OnNewMail(MOOSMSG_LIST &NewMail);
    //where we do the work
    bool Iterate();
    //called when we connect to the server
    bool OnConnectToServer();
    //called when we are starting up..
    bool OnStartUp();



    //state our interest in variables
    void DoRegistrations();

    //we'll call this if/when we receive a vehicle status message
    bool OnVehicleStatus(CMOOSMsg & Msg);

    //we'll call this if/when we receive a heading message
    bool OnHeading(CMOOSMsg & Msg);


};

#endif
```

Listing 5: Simplest Application - main.cpp

```
#include "SimpleApp.h"

#include <MOOSGenLib/MOOSGenLibGlobalHelper.h>

//default constructor
CSimpleApp::CSimpleApp()
{
}
//default (virtual) destructor
CSimpleApp::~CSimpleApp()
{
}

/**
Called by base class whenever new mail has arrived.
Place your code for handling mail (notifications that something
has changed in the MOOSDB in this function

Parameters:
    NewMail :  std::list<CMOOSMsg> reference

Return values:
    return true if everything went OK
    return false if there was problem
**/
bool CSimpleApp::OnNewMail(MOOSMSG_LIST &NewMail)
```

```
{
    MOOSMSG_LIST::iterator p;

    for(p=NewMail.begin();p!=NewMail.end();p++)
    {
    //lets get a reference to the Message - no need for pointless copy
    CMOOSMsg & rMsg = *p;

    // repetitive "ifs" is one way to build a switch yard for
    // the messages
    if(MOOSStrCmp(rMsg.GetKey(),"VehicleStatus"))
    {
        //this message is about something called "VariableX"
        OnVehicleStatus(rMsg);
    }
    else if(MOOSStrCmp(rMsg.GetKey(),"Heading"))
    {
        //this message is about something called "VariableY"
        OnHeading(rMsg);
    }
    }



    return true;
}


/**
called by the base class when the application has made contact with
the MOOSDB and a channel has been opened. Place code to specify what
notifications you want to receive here.
**/
bool CSimpleApp::OnConnectToServer()
{
    //do registrations
    DoRegistrations();

    return true;
}

/** Called by the base class periodically. This is where you place code
which does the work of the application **/
bool CSimpleApp::Iterate()
{

    return true;
}

/** called by the base class before the first ::Iterate is called. Place
startup code here - especially code which reads configuration data from the
mission file **/
bool CSimpleApp::OnStartUp()
{
    //do registrations - its good practice to do this BOTH in OnStartUp and
    //in OnConnectToServer - that way if comms is lost registrations will be
    //reinstigated when the connection is remade
    DoRegistrations();

    return true;
}
```

```
bool CSimpleApp::OnVehicleStatus(CMOOSMsg & Msg)
{
    MOOSTrace("I (%s) received a notification about \"%s\" the details are:\n",
          GetAppName().c_str(),
          Msg.GetKey().c_str());

    //if you want to see all the details you can print a message...
    //Msg.Trace();

    if(!Msg.IsString())
    return MOOSFail("Ouch - I was promised \"VehicleStatus\" would be a string!")
        ;

    //OK the guy who wrote the program that publishes VehicleStatus wrote me an
    //email saying the format of the message is:
    //Status = [Good/Bad/Sunk], BatteryVoltage = <double>, Bilge=[on/off]
    //so here we parse the bits we want from the string
    std::string sStatus="Unknown";
    if(!MOOSValFromString(sStatus,Msg.GetString(),"Status"))
    MOOSTrace("warning field \"Status\" not found in VehicleStatus string %s",
        MOOSHERE);

    double dfBatteryVoltage=-1;
    if(!MOOSValFromString(dfBatteryVoltage,Msg.GetString(),"BatteryVoltage"))
    MOOSTrace("warning field \"BatteryVoltage\" not found in VehicleStatus string
        %s",MOOSHERE);

    //simple print out our findings..
    MOOSTrace("Status is \"%s\" and battery voltage is %.2fV\n",sStatus.c_str(),
        dfBatteryVoltage);

    return true;
}


bool CSimpleApp::OnHeading(CMOOSMsg & Msg)
{
    MOOSTrace("I (%s) received a notification about \"%s\" the details are:\n",
          GetAppName().c_str(),  //note GetAppName() returns the name of this
              application as seen by the DB
          Msg.GetKey().c_str()); //note GetKey() return the name of the variable


    //if you want to see all the details you can print a message...
    //Msg.Trace();

    //you might want to be sure that the message is in the format you were
        expecting
    //in this case heading comes as a single double...

    if(!Msg.IsDouble())
    return MOOSFail("Ouch - was promised \"Heading\" would be a double %s",
        MOOSHERE);

    double dfHeading = Msg.GetDouble();
    double dfTime = Msg.GetTime();

    MOOSTrace("The heading (according to process %s),at time %f (%f since
        appstart) is %f\n",
          Msg.GetSource().c_str(), //who wrote it
          dfTime,//when
```

```
            dfTime-GetAppStartTime(),//time since we started running (easier to
                read)
            dfHeading);//the actual heading

    return true;
}



void CSimpleApp::DoRegistrations()
{
    //register to be told about every change (write) to "VehicleStatus"
    m_Comms.Register("VehicleStatus",0);

    //register to be told about changes (writes) to "Heading" at at most
    //4 times a second
    m_Comms.Register("Heading",0.25);

    return;
}
```

4.1 Registering for Mail

An instance of `MOOSApp` comes with an `CMOOSCommClient` object called `m_Comms` — this is the guy that allows us to register for mail with a call to `m_Comms.Register()` . In Listing 5 two such calls are made in a function called `DoRegistrations` where we register for messages (mail) about "Heading" and "VehicleStatus". Note that the former will be delivered at a maximum of 4Hz (irrespective of how often some unknown external process is writing the data [6]) while "VehicleStatus" messages will be delivered to us every time someone writes "VehicleStatus".

4.1.1 Where and When Should the Registrations Occur?

I advise folk to register for mail in two places. Once at the end of `OnStartup()` and once in `OnConnectToServer()` . The reasons for this are as follows.

- It is usual to execute code in `OnStartUp` which determines what mail we want to register for (e.g. as a result of parsing the mission file which should happen in `OnStartUp` — see later for more on Mission Files.)
- Connection to the DB is asynchronous (it depends on what else is going on in the network). Accordingly `OnConnectToServer` might be called before or after `OnStartUp()` so in the former case we'd want to perform registrations at the end of `OnStartUp` and in the latter case in `OnConnectToServer` . [7]

[6] This is a good thing - it stops some over-zealous process (which you didn't write) causing copious amounts of mail to your door.

[7] Hmmm, upon reflection maybe there is a case to be made for having persistent registrations so the CommsClient remembers all the registrations you ever ask for and makes sure that these are preserved across DB connections/disconnections...I would welcome a view on this...

4.2 Parsing Messages

Perhaps the most important thing to focus on in Listing 5 is the `OnNewMail` method. Here you can see how (in this example) the list of `CMOOSMsgs` that have been delivered to us (in response to earlier calls to `m_Comms.Register` - see section 4.1) are cracked and execution is routed as a function of the variable name which each message pertains to. Pretty simple.

So, having marshalled execution to regions of code dedicated to handling `CMOOSMsgs` pertaining to particular named data (like heading of vehicle status in our rather contrived example) we need to extract the data itself. `CMOOSMsgs` can contain double-data (`Msg.GetDouble()`) or string-data (`Msg.GetString()`). In this case we were promised by other developers that "VehicleStatus" will be a string-variable and "Heading" will be a double-variable. You can see this contract being checked in the two message cracking methods in Listing 5. Extracting the double precision data from the heading messages is trivial, however extracting data from the string of vehicle status is more interesting and is a common problem in MOOSApplications. As you might expect, there are a whole load of tools ready and waiting to help you with this task (many of them are found in the header file `MOOSGenLibGlobalHelper.h` — see Section 10). We'll talk about string manipulation in the context of Parsing `CMOOSMsgs` in Section 4.5.

4.3 Using `::PeekMail` for Sorting Mail

The example code in Listing 5 used a bunch of "if" statements inside an interation overall message to marshall the incoming message to the correct handler. It is possible to do away with writing `for` loop by using the `CMOOSCommsClient::PeekMail` function [8]. This function is passed a reference to the whole list of incoming messages and extracts a particular message according to the name of the variable we are interested in. **Importantly** this method can extract the most recent message of a given name. Why is this useful? Well, imagine you've requested to receive notifications about *every* write to a named variable. It is quite possible that some other client published data about that particular variable many times since our application last received mail. Hence we will receive multiple messages pertaining to the same variable in our `MOOSMSG_LIST` when `OnNewMail` is called. We can imagine that it would be pretty useful at times to only act on the most uptodate (recent) message. A call to `PeekMail` can also remove (rather than copy) the message from the `MOOSMSG_LIST` . Look at Listing 6 for a coded example.

4.4 Checking for Stale Messages

When a process starts up and registers for mail, it has no knowledge about the state of variables stored in the DB. If at the time an application registers for notifications about a particular variable that variable already exists in the `MOOSDB` it will be sent a message about that variable which will appear in the mail on the first time `OnNewMail` is invoked. This might mean that an application receives a message from the DB about a posting which is days old. An obvious approach would be to check the time field of each message and only

[8] i.e you would call `m_Comms.PeekMail(...)`

process messages that are "recent". The method `CMOOSMsg::IsSkewed` is an easy way to check this - behind the scenes it simply makes sure the message in question has a time field within a few seconds of the current time. Of course one may want better precision on what counts as stale mail and want to write your own method to do so. However `IsSkewed` is ofen useful to make the first cut. Look at Listing 6 for a coded example.

Listing 6: An alternative way of handling mail. This time using `PeekMail` and also checking for stale messages.

```
/** an alternative OnNewMail using PeekMail and checking for stale
    messages */
bool CSimpleApp::OnNewMail(MOOSMSG_LIST &NewMail)
{
    CMOOSMsg Msg;
    double dfNow = MOOSTime();
    if(m_Comms.PeekMail(NewMail,"VehicleStatus",Msg,false,true))
    {
    if(!Msg.IsSkewed(dfNow))
    {
        OnVehicleStatus(Msg);
    }
    }
    if(m_Comms.PeekMail(NewMail,"Heading",Msg,false,true))
    {
    if(!Msg.IsSkewed(dfNow))
    {
        OnHeading(Msg);
    }
    }

    return true;
}
```

4.5 Parsing Strings

The developer is in no way obligated to use the string parsing methods provided by the MOOS Core libraries. But they are there to be used and by doing so the developer is more likely to adopt string structures/formats used by the multitude of MOOS processes already in existence.

The good thing about sending string data is that multiple fields can be sent at the same time. Typically a string is sent as a comma separated list of parameter=value pairs where value is itself a string, double or string representation of a matrix. For example, the developer of a process which publishes "VehicleStatus" messages told us that the data would use the string field of MOOSMsgs and would contain the following fields:

Status one of "good", "bad" or "sunk" (the latter presumably being a special case of "bad")

BatteryVoltage a double value

Bilge a string, one of "On" or "Off'.

The data corresponding to each of these tokens can be extracted from the string of the `CMOOSMsg` (or any string for that matter) by using the

::MOOSValFromString family of functions found in MOOSGenLib — see Section 10. The reader's attention is also drawn to the way in which chunks of numerical (std::vector<double>) data can be sent and extracted from strings.

4.6 Testing The New Application

So we've built our new application — how do we test it? Well, usually you'd run it in the system in which it was designed to reside, but this is a fictitious system and no processes that write "VehicleStatus' and or "Heading" exist. So we could either write new processes that do, or use a preexisting tool to write data to the MOOSDB and see our application respond appropriately. The former is described in Section 4.7 and the latter in Section 4.6.1.

4.6.1 Testing with uMS

The simplest way to test our example is to launch the graphical tool uMS [9]. The steps are as follows:

1. Start an instance of MOOSDB (either double click in Windows or type MOOSDB & from a terminal in linux).
2. Start an instance of uMS . Press the connect button (the default is to connect to a MOOSDB using the MOOSDB defaults settings). You should see uMS come to life.
3. Start an instance of your new application. In this case it is called Ex2. You should see its existence being detected by both the MOOSDB and uMS .
4. Pick a blank line in uMS and ctrl-left click in the left most column. From here you can "Poke the MOOS" with new data (indeed clicking on an existing variable allows you to change that variable).
5. Start by poking "Heading" into the system as a double. You should see your application (Ex2) print to the screen that it has received heading data.

Figure 4.6.1 shows a screen shot of this process taken from my machine (which today happens to be a linux box).[10]

4.7 Testing Ex2 with Another Application

A more interesting way of testing our new application is to write another new process which writes "VehicleStatus" and "Heading"[11]. The code for such a process can be found in the Ex3 directory. The main thing to notice is that the ::Iterate() function is now populated with code which publishes data using

[9] Again, it is assumed that you have already built the MOOS distribution which includes uMS which uses the FLTK library. See the sibling document on Graphical tools for more information on uMS .

[10] The screen shot isn't brilliant - you may have to zoom in or look at the raw image (UsinguMS.eps) if you want real detail.

[11] We could write two - one for Heading and one for VehicleStatus but there is nothing new to be learned there.

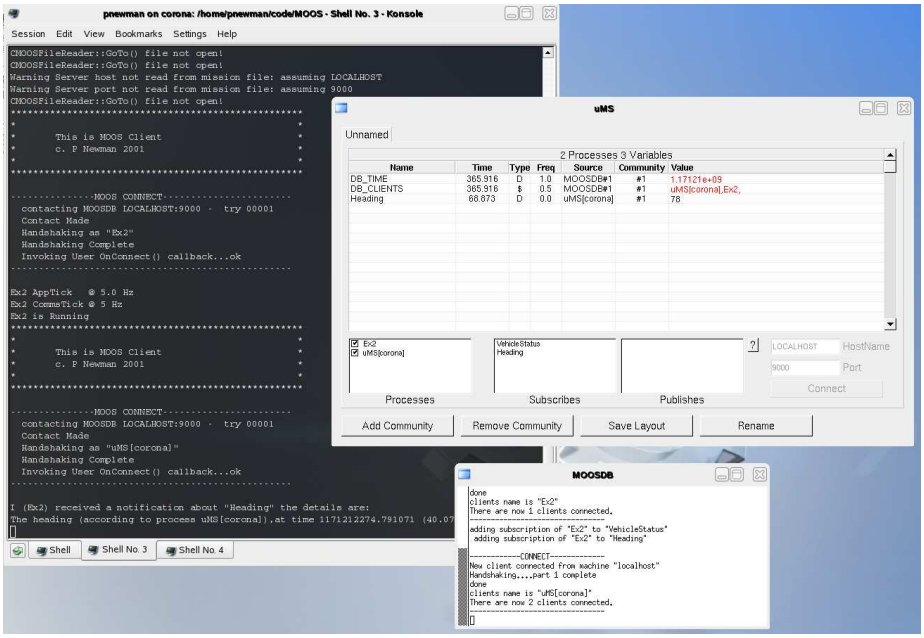

Figure 2: Using uMS to poke data into the `MOOSDB` to test a new application.

the `Notify` method of the `m_Comms` member. The guts of the new application (simulator) is given in Listing 7. Of course the details of how the simulator works are irrelevant (I'd hardly call it a simulator); the code is provided here just to show how data is posted to the DB. Note that the third field to the Notify method is optional - if you don't supply a time, MOOSTime() is called behind the scenes for you [12].

Listing 7: Code for a "heading and status" simulator

```
#include "Simulator.h"
#include <math.h>

//default constructor
CSimulator::CSimulator()
{
}

//default (virtual) destructor
CSimulator::~CSimulator()
{
}

/**
Called by base class whenever new mail has arrived.
Place your code for handling mail (notifications that something
has changed in the MOOSDB in this function

Parameters:
```

[12] It is common practice when sending string data to send the time field in the string as well, but it is not a requirement of course – you can send what you want in strings.

```
    NewMail :  std::list<CMOOSMsg> reference

Return values:
    return true if everything went OK
    return false if there was problem
**/
bool CSimulator::OnNewMail(MOOSMSG_LIST &NewMail)
{
    return true;
}

/**
called by the base class when the application has made contact with
the MOOSDB and a channel has been opened. Place code to specify what
notifications you want to receive here.
**/
bool CSimulator::OnConnectToServer()
{
    return true;
}

/** Called by the base class periodically. This is where you place code
which does the work of the application **/
bool CSimulator::Iterate()
{
    static int k = 0;
    if(k++%10==0)
    {
    //simulate some brownian motion
    static double dfHeading = 0;
    dfHeading+=MOOSWhiteNoise(0.1);

    //publish the data (2nd param is a double so it will be forever double data
        ...)
    m_Comms.Notify("Heading",dfHeading,MOOSTime());
    }
    if(k%35==0)
    {

    static double dfVolts  = 100;
    dfVolts-=fabs(MOOSWhiteNoise(0.1));
    std::string sStatus = MOOSFormat("Status=%s,BatteryVoltage=%.2f,Bilge␣=␣%s",
                    dfVolts>50.0? "Good":"Bad",
                    dfVolts,
                    k%100>50?"On":"Off");

    //publish the data (2nd param is a std::string so it will be forever string
        data...)
    m_Comms.Notify("VehicleStatus",sStatus,MOOSTime());
    }
    return true;
}

/** called by the base class before the first ::Iterate is called. Place
startup code here - especially code which reads configuration data from the
mission file **/
bool CSimulator::OnStartUp()
{
    return true;
}
```

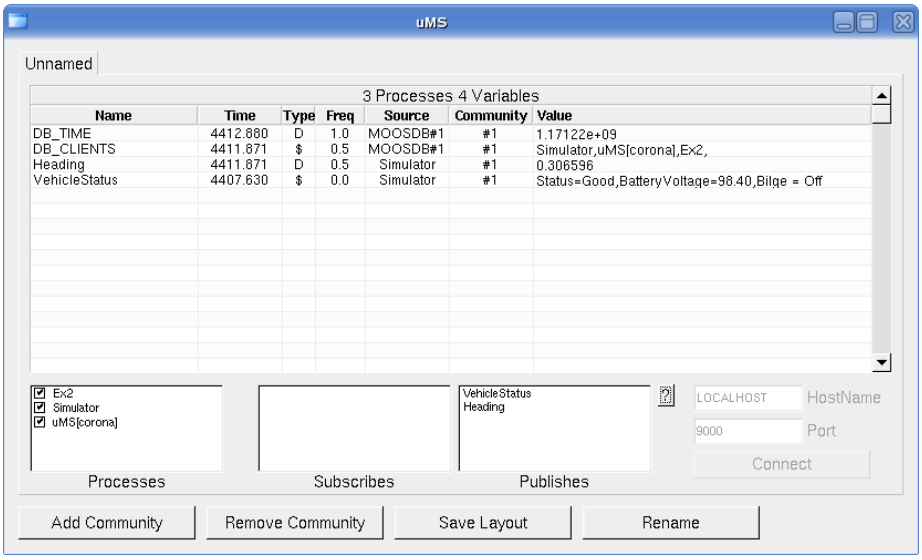

Figure 3: Spying on Ex2 and the crude simulator (Ex3) in action

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)
Source Community	The community to which the source process belongs — see the Section on "MOOS C

Table 1: Contents of MOOS Message

So after building Ex3 you can start it up and see Ex2 respond to the messages being posted. You should also get warm fuzzies about seeing the two applications in action via `uMS` . Figure 4.7 is a screen shot of uMS running when Ex3 (which registers with the DB as Simulator) and Ex2 are running.

5 Message Content — `CMOOSMsg`

The communications API in MOOS allows data to be transmitted between `MOOSDB` and 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 contain other salient information as 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[13]. 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 data. Irrespective of this, experience has shown that the benefits of using strings far outweigh the difficulties. In particular:

- Strings are human-readable – debugging is trivial, especially using a tool like MOOSScope. (see the document on Graphical tools for more information.)
- 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 Important `CMOOSApp` Methods

6.1 Reading Configurations From File

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 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`

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

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
}
```

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

Each statement is written as

```
ParameterName = Value
```

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

Figure 6.1 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 — in this case (`CMOOSInstrument` and `CMOOSApp`).

6.2 Parsing Configuration Blocks

The library MOOSGenLin (see section 10) contains many functions and classes designed to help with parsing mission files. In particular the `CMOOSApp` class comes equipped with its own `CMOOSProcessConfiguration` object called `m_MissionReader` . By the time OnStartup is called, this object is already configured for use (i.e. it already knows which block it should be reading in the config file) and can be queried at will. Listing 8 shows a modified version of our simulator (found in the "Ex4" directory). Note the new code appearing in `OnStartup` which looks for parameters in the mission file by making calls on the ProcessConfig reader object. You might also want to refer back to Sections 4.5 and 10 to review the methods available for string passing.

Listing 8: A modified simulator application which reads configuration information from its mission file.

```
#include "Simulator.h"
#include <math.h>

//default constructor
CSimulator::CSimulator()
{
}

//default (virtual) destructor
CSimulator::~CSimulator()
{
}

/**
Called by base class whenever new mail has arrived.
Place your code for handling mail (notifications that something
has changed in the MOOSDB in this function

Parameters:
    NewMail :  std::list<CMOOSMsg> reference

Return values:
    return true if everything went OK
    return false if there was problem
**/
bool CSimulator::OnNewMail(MOOSMSG_LIST &NewMail)
{
    return true;
}

/**
called by the base class when the application has made contact with
the MOOSDB and a channel has been opened. Place code to specify what
notifications you want to receive here.
**/
bool CSimulator::OnConnectToServer()
{
    return true;
}

/** Called by the base class periodically. This is where you place code
which does the work of the application **/
bool CSimulator::Iterate()
{
    static int k = 0; // a simple counter to simulate a simulator - not an
        important detail...
    if(k++%10==0)
    {
    //simulate some brownian motion
    m_dfHeading+=MOOSWhiteNoise(0.1);

    //publish the data (2nd param is a double so it will be forever double data
        ...)
    std::string sVarName = m_sVehicleName+"_Heading";
    m_Comms.Notify(sVarName,m_dfHeading,MOOSTime());
    }
    if(k%35==0)
    {
```

```
    m_dfBatteryVoltage-=fabs(MOOSWhiteNoise(0.1));
    std::string sStatus = MOOSFormat("Status=%s,BatteryVoltage=%.2f,Bilge=%s",
                        m_dfBatteryVoltage>50.0? "Good":"Bad",
                        m_dfBatteryVoltage,
                        m_sBilge.c_str());

    //publish the data (2nd param is a std::string so it will be forever string
        data...)
    //note how name of variable is set by what was read from configuration file
    std::string sVarName = m_sVehicleName+"_Status";
    m_Comms.Notify(m_sVehicleName,sStatus,MOOSTime());
    }
    return true;
}

/** called by the base class before the first ::Iterate is called. Place
startup code here - especially code which reads configuration data from the
mission file **/
bool CSimulator::OnStartUp()
{
    //here we extract the vehicle name..
    m_sVehicleName = "UnNamed";
    if(!m_MissionReader.GetConfigurationParam("VehicleName",m_sVehicleName))
    MOOSTrace("Warning parameter \"VechicleName\" not specified. Using default of
        \"%s\"\n",m_sVehicleName.c_str());


    //here we extract a vector of doubles from the configuration file
    std::vector<double> vInitialLocation(3,0.0);
    int nRows=vInitialLocation.size();
    int nCols = 1;
    if(!m_MissionReader.GetConfigurationParam("InitialLocation",vInitialLocation,
        nRows,nCols))
    MOOSTrace("Warning parameter \"InitialLocation\" not specified. Using default
        of \"%s\"\n",DoubleVector2String(vInitialLocation).c_str());

    //here we extract a more complicated compound string parameter
    std::string sComplex;
    if(m_MissionReader.GetConfigurationParam("InitialConditions",sComplex))
    {
    //OK now we can suck out individual parameters from sComplex

    //what is the initial Bilge condition status?
    m_sBilge = "Off";
    MOOSValFromString(m_sBilge,sComplex,"Bilge");

    //what is the initial battery Voltage?
    m_dfBatteryVoltage = 100.0;
    MOOSValFromString(m_dfBatteryVoltage,sComplex,"BatteryVoltage");

    //what is the initial heading
    m_dfHeading = 0;
    MOOSValFromString(m_dfHeading,sComplex,"Heading");

    }
    else
    {
    //bad news - this one is compulsory for this application...
    return MOOSFail("no \"InitialConditions\" specified in mission file (
        compulsory)\n");
    }
```

```
    MOOSTrace("Verbose Summary:\n");
    MOOSTrace("\tVehicle is called : %s\n",m_sVehicleName.c_str());
    MOOSTrace("\tInitial Location is  : %s\n",DoubleVector2String(
        vInitialLocation).c_str());
    MOOSTrace("\tHeading is  : %f\n",m_dfHeading);
    MOOSTrace("\tBatteryVoltage is  : %s\n",m_sBilge.c_str());



    return true;
}
```

We can now run up the simulator (which compiles using the CMake files supplied to "Ex4") and pass it a mission file as a parameter. But first we must create a suitable mission file — for example "Ex4.moos" which is reproduced in Listing 9.

Listing 9: A simple mission file for Ex4

```
//tell all processes where the DB is (default is localhost:9000)
ServerPort = 9000
Serverhost = localhost


ProcessConfig = Simulator
{
        //how fast should iterate be called? (used by CMOOSApp)
        AppTick= 10

        //how responsive should comms be? (used by CommsClient)
        CommsTick = 10

        //name of the vehicle
        VehicleName = TheGoodShipMOOS

        //initial location
        InitialLocation = [3x1]{0,1,2}

        //a more complex compound config string
        InitialConditions = Bilge=off, BatteryVoltage=101, Heading  = 0.57

}
```

In Figure 5 you can see a screen shot of what you should see happening when you launch `Ex4` and point it at `Ex4.moos` (don't forget to launch a `MOOSDB` first[14].

6.3 The Role of `AppTick` and `CommsTick`

Every configuration block can set the `AppTick` and `CommsTick` properties of a `CMOOSApp` derived application — see for example Figure 6.1. The former specifies the target rate (in Hz and can be less that 1.0) that `::Iterate` will be called at. Setting `AppTick` to zero is a special case and causes Iterate to be called as fast as possible (in other words the `::Run` method of CMOOSApp loops without any "sleep" period) — use this with caution and good manners.

[14] I tend to always have one running on my machine.

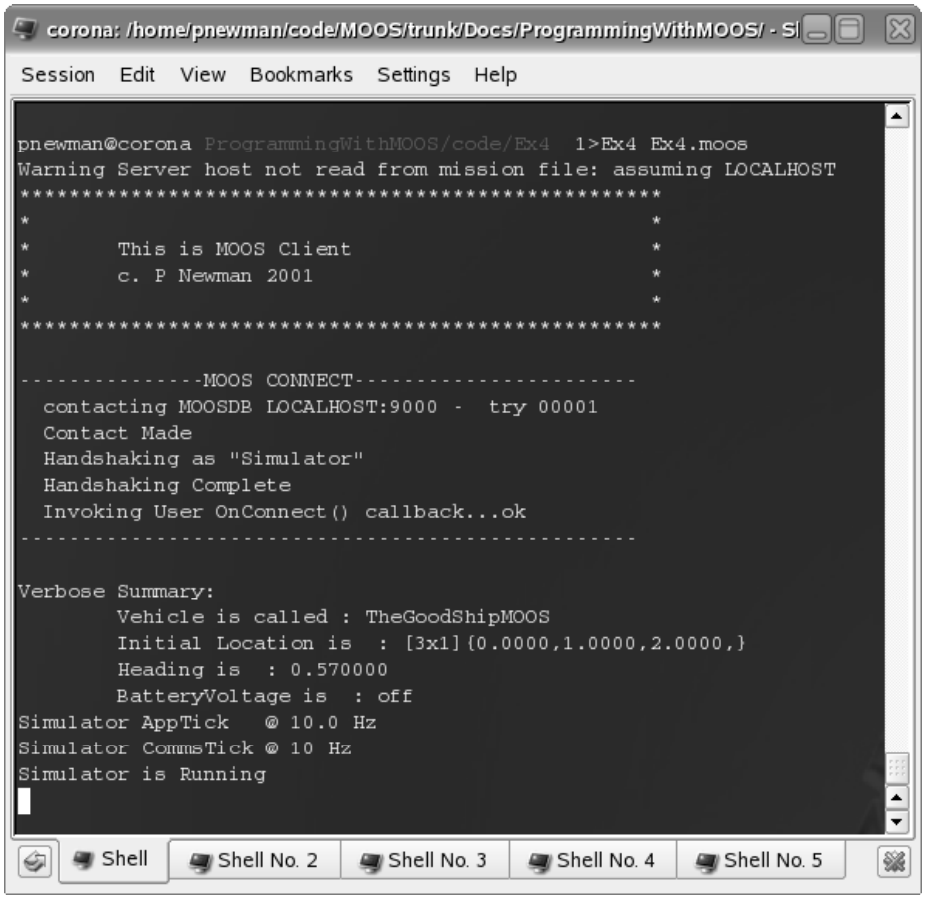

Figure 5: A screen shot of `Ex4` being run. Note how it was launched and has picked up the configuration parameters from Ex4.moos (see Listing 9)
.

The `CommsTick` variable dictates how often the `m_Comms` object owned by every `CMOOSApp` contacts the DB in a never-ending quest to collect and post data from and to the DB. High values will lead to snappy response times if combined with high `AppTick` . If your application needs to call Iterate to do work often but you don't expect or require fast communication with other processes, then a high AppTick and a low CommsTick will suffice.

7 Using the `CMOOSVariables` with `CMOOSApp`

There is one other functionality provided by `CMOOSApp` which can prove very useful and that is the ability to create and manage runtime variables. You may well find that your application needs to maintain a representation of system state using a whole set of variables which are set according to the contents of incoming mail. To be concrete (and perhaps a bit naive) one can imagine a navigation application possessing variables for heading, speed, fuel, engine speed, headwind, current, depth. Each of these variables would appear in the mail processing switch yard (where they are updated to the values contained in individual MOOS messages) and each variable would have to be persistent — in a object-oriented outlook this would mean a whole list of member variables. For a few such variables this is no big deal but in applications that have requested notifications on large numbers of variables, the mail processing function becomes long and tedious and the class itself becomes peppered with simple member variables.

`CMOOSApp` offers a way to soothe this frustration with the following functions (taken from `MOOSApp.h`).

```
    ////////////////////////////////////////////////
    //  DYNAMIC VARIABLES - AN OPTIONAL GARNISH
    ////////////////////////////////////////////////

    /** Add a dynamic (run time) variable
        @param sName name of the variable
        @param sSubscribeName if you call RegisterMOOSVariables() the
            variable will be updated with mailcalled <sSubscribeName> if
            and when you call UpdateMOOSVariables()
        @param sPublishName  if you call PublishFreshMOOSVariables() (and
            you've written to the dynamic variable since the last call)
            the variable will be published under this name.
        @param CommsTime - if sSubscribeName is not empty this is the
            minimum time between updates which you are interested in
            knowing about, so if CommsTime=0.1 then the maximum update
            rate you will see on the variable from the DB is 10HZ.  */
    bool AddMOOSVariable(std::string sName,std::string sSubscribeName,std
        ::string sPublishName,double dfCommsTime);

    /** return a pointer to a named variable */
    CMOOSVariable * GetMOOSVar(std::string sName);

    /** Register with the DB to be mailed about any changes to any dynamic
         variables which were created with non-empty sSubscribeName fields
         */
```

```
    bool RegisterMOOSVariables();

    /** Pass mail (usually collected in OnNewMail) to the set of dynamic
        variables. If they are interested (mail name matches their
        subscribe name) they will update themselves automatically */
    bool UpdateMOOSVariables(MOOSMSG_LIST & NewMail);

    /** Set  value in a dynamic variable if the variable is of type double
         (type is set on first write )*/
    bool SetMOOSVar(const std::string & sName,const std::string & sVal,
        double dfTime);

    /** Set  value in a dynamic variable if the variable is of type string
         (type is set on first write ) */
    bool SetMOOSVar(const std::string & sVarName,double dfVal,double
        dfTime);

  /** Send any variables (under their sPublishName see AddMOOSVariable)
      which been written to since the last call of
      PublishFreshMOOSVariables()*/
    bool PublishFreshMOOSVariables();
```

The idea is that with a call to `AddMOOSVariable` one can dynamically create a named variable (which behind the scenes is of type `CMOOSVariable`). As you do so, you specify the name of the messages (the string returned by calls to `CMOOSMsg.GetKey()`) which should be used to update this variable and also the name under which you wish to publish this data should you wish to undertake a notification. A concrete case may clarify things. Consider the case of a GPS sensor application, calling `AddMOOSVariable("X","","GPS_X",0)` will create a dynamic variable called "X". I can set the value of this variable, presumably after successful parsing of a string read from a serial port) via `SetMOOSVar("X",...)` and, should I desire, retrieve it via `GetMOOSVar("X",...)`. I don't need to have a `m_dfGPSX` variable explicitly declared in any class - it is made at run time. Now a role of this hypothetical application is to inform the MOOS community about the vehicle location — we need to do a "notify" on "GPS_X". This is easily achieved by calling `PublishFreshMOOSVariables` which calls a "notify" on any dynamic variable which has been written to since the last invocation (so in this case if no new GPS data had been received from the sensor no new data would be published to the MOOSDB). Imagine now you have ten things you might like to post to the database as and when occasion dictates — a single call to `PublishFreshMOOSVariables` handles the whole thing for you.

Finally, consider the symmetrical case where instead of pushing data out we want to read data in. In this case we would do something like `AddMOOSVariable("Temp","ENGINE_TEMP","",(`. Here we are making a local MOOSVariable called `Temp` and telling the application (which is a `MOOSApp`) that it is a mirror of the `MOOSDB` code variable called `ENGINE_TEMP`. So instead of having a `if` statement like

```
if(Msg.Key()=="ENGINE\_TEMP")
{
}
```

in the OnNewMail we can simply call UpdateMOOSVariables(NewMail) and if within the list of MOOSMsg there is a message pertaining to ENGINE_TEMP it will be automatically used to update the Temp variable. Note you do need to make a call to RegisterMOOSVariables() to make sure that your application does all the registrations for all your MOOSvariables for you.

8 Getting By Without MOOSApp

There may be times when a developer does not wish to write an application from scratch using CMOOSApp , preferring to add the MOOS communications functionality to an existing application. This is an easy thing to achieve; a typical plan is laid out in the following five steps:

1. Instantiate a persistent instance of CMOOSCommClient – perhaps as class member or even as a global singleton.
2. If required, use the CMOOSCommClient::SetOnConnectCallBack and CMOOSCommClient::SetOnDisConnectCallBack methods to tell the communications object what functions to call when a connection is made (or lost) with the MOOSDB . Note that these callbacks will happen in a separate thread. The latter of these callback registration functions is rarely used but symmetry is attractive.
3. Call the non-blocking CMOOSCommClient::Run passing the name (or IP address) of the machine hosting the DB, the port on which it is listening (usually 9000 but that can be configured) and the rate at which you want the communications thread to run in Hz (the default is 5Hz).
4. As soon as calls to CMOOSCommClient::IsConnected start returning true, you are free to start registering for notifications and posting your own data, as described in earlier sections. It is a good plan to put your registration code in the connection callback.
5. In your existing code periodically (perhaps via a timer in a gui application) call CMOOSCommClient::Fetch to retrieve whatever mail has been delivered to your application since the last invocation of CMOOSCommClient::Fetch (the comms object will have been having regular chats with the MOOSDB in the background while your own code has been doing its thing).

9 Other Bells and Whistles

In releases post 7.0.1 several new methods are available via CMOOSApp .

9.1 Process Status Summaries

Every few seconds CMOOSApp publishes a status string. If the MOOS name of a process is XYZ then a status string is published under the name XYZ_STATUS . By default the status string is formatted by the virtual member function std::string CMOOSApp::MakeStatusString() which formats a string containing:

- process uptime
- names of all messages published so far
- names of messages currently subscribed to.

By overloading `std::string CMOOSApp::MakeStatusString()` you can append or replace the status string with whatever you choose — `CMOOSApp` will call your version in preference to its own.

9.2 Automatic Handling of <PROCESSNAME>_CMD Messages

It is commonplace to want to have processes monitoring variables which contain instructions on how to behave or which request certain actions — one could invoke the umbrella term "command messages". Post release 7.0.1, `CMOOSApp` contains some plumbing to manage the handling of such messages. By extending the API it also helps best practice of using a common message naming policy for all such messages. By calling `CMOOSApp::EnableCommandMessageFiltering(true)` (for example in `OnStartup`) CMOOSApp will, behind the scenes, peruse incoming mail for messages called `XYZ_CMD` where the XYZ is a **capitalised** MOOS Community name of process (the name with which process registers itself with the DB). If any such messages are found, the virtual function `CMOOSApp::OnCommandMsg(CMOOSMsg Msg)` is called. The default implementation does nothing — overload this function to perform your own customised processing.

The command message filtering facility can also be turned on in any process's configuration block by adding the line `CatchCommandMessages = true` . By default command message filtering is off.

10 MOOSGenLib Functions

The library MOOSGenLib provides a plethora of functions that tend to be useful for programs using MOOS. They are quite self explanatory and are best perused by looking at the `MOOSGenLibGlobalHelper.h` header file which is included here for your delectation.

Listing 10: MOOSGenLibGlobalHelper.h - many goodies live here

```
//////////////////////////////////////////////////////////////////////
//
//   MOOS - Mission Oriented Operating Suite
//
//   A suit of Applications and Libraries for Mobile Robotics Research
//   Copyright (C) 2001-2005 Massachusetts Institute of Technology and
//   Oxford University.
//
//   This software was written by Paul Newman at MIT 2001-2002 and Oxford
//   University 2003-2005. email: pnewman@robots.ox.ac.uk.
//
//   This file is part of a  MOOS Core Component.
//
//   This program is free software; you can redistribute it and/or
//   modify it under the terms of the GNU General Public License as
//   published by the Free Software Foundation; either version 2 of the
//   License, or (at your option) any later version.
//
```

```
//   This program is distributed in the hope that it will be useful,
//   but WITHOUT ANY WARRANTY; without even the implied warranty of
//   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
//   General Public License for more details.
//
//   You should have received a copy of the GNU General Public License
//   along with this program; if not, write to the Free Software
//   Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
//   02111-1307, USA.
//
//////////////////////////     END_GPL    ////////////////////////////////////
/*! \file MOOSGenLibGlobalHelper.h */


#ifndef MOOSGenLibGlobalHelperh
#define MOOSGenLibGlobalHelperh


#ifndef PI
#define PI 3.141592653589
#endif

#include <string>
#include <list>
#include <vector>
#include <sstream>
#include <algorithm>




////////////////////// STRING MANIPULATION/EXTRATION TOOLS
    ////////////////////////////////

typedef std::list<std::string> STRING_LIST ;

//find the location of sToken in a string sSource with or without case
    sensitivity
size_t  MOOSStrFind( const std::string &sSource , const std::string & sToken,bool
     bInsensitive=false);


//following function finds token = value in a list of such strings
bool MOOSGetValueFromToken(STRING_LIST & sParams,const std::string & sToken,std::
    string & sVal);

//following functions extract a value from a string containing comma seperated
    pairs of Token = Val....
//by default case sensitivity on the Token is off. Thus  given two strings S1="X=
    AbCd" and S2="x=AbCd",
//invoking  MOOSValFromString(s,S1,"x",true) and MOOSValFromString(s,S2,"x",true)
     with both insert "AbCd" into s
bool MOOSValFromString(std::string & sVal,const std::string & sStr,const std::
    string & sTk,bool bInsensitive=false);
bool MOOSValFromString(double & dfVal,const std::string & sStr,const std::string
    & sTk,bool bInsensitive=false);
bool MOOSValFromString(float  & fVal,const std::string & sStr,const std::string
    & sTk,bool bInsensitive=false);
bool MOOSValFromString(long   & nVal,const std::string & sStr,const std::string
    & sTk,bool bInsensitive=false);
bool MOOSValFromString(int    & nVal,const std::string & sStr,const std::string
    & sTk,bool bInsensitive=false);
```

```
bool MOOSValFromString(bool   &  bVal,const std::string & sStr,const std::string
    & sTk,bool bInsensitive=false);
bool MOOSValFromString(unsigned int &  nVal,const std::string & sStr,const std::
    string & sTk,bool bInsensitive=false);
bool MOOSValFromString(std::vector<double> &dfValVec,int &nRows,int &nCols, const
     std::string & sStr, const std::string & sToken,bool bInsensitive=false);
bool MOOSValFromString(std::vector<unsigned int> &nValVec, int &nRows, int &nCols
    , const std::string & sStr, const std::string & sToken,bool bInsensitive=
    false);

//the following simply parse a MOOSFormated vector [nxm]{a,b,c...}
bool MOOSVectorFromString(const std::string & sStr,std::vector<double> & dfVecVal
    ,int & nRows,int & nCols);
bool MOOSVectorFromString(const std::string & sStr,std::vector<float> & fValVec,
    int & nRows, int & nCols);
bool MOOSVectorFromString(const std::string & sStr,std::vector<unsigned int> &
    dfVecVal,int & nRows,int & nCols);



/** write a std::vector<double> to a string (using MOOS Notation) */
std::string DoubleVector2String(const std::vector<double> & V);

/** write a std::vector<double> to a stringstream (using MOOS Notation)*/
std::stringstream & Write (std::stringstream & os,const std::vector<double> & Vec
    );

/** write a std::vector<int> to a stringstream (using MOOS Notation)*/
std::stringstream & Write (std::stringstream & os,const std::vector<int> & Vec);

//the ubiquitous chomp function
std::string MOOSChomp(std::string &sStr, const std::string &sTk=",",bool
    bInsensitive=false);

/** remove all characters in sTok from sStr*/
void MOOSRemoveChars(std::string & sStr,const std::string & sTok);

/** convert string to upper case*/
void MOOSToUpper(std::string &str);

/** remove white space form start and end of a string */
void MOOSTrimWhiteSpace(std::string & str);

/**returbn true if numeric */
bool MOOSIsNumeric(std::string  str);

/** case insensitive string comparison. returns true if equal */
bool MOOSStrCmp(std::string s1,std::string s2);

/** pattern matching using * and ?. returns true if sPattern matches sString */
bool MOOSWildCmp(const std::string & sPattern, const std::string & sString ) ;


////////////////////// TIMING TOOLS  ////////////////////////////////

/**generic timing functions*/
double GetMOOSSkew();
void SetMOOSSkew(double dfSkew);

/** return the offset between DB time and client time*/
double GetMOOSSkew();
```

```
/** set the rate at which time is accelerated (from start of unix time) */
bool SetMOOSTimeWarp(double dfWarp);

/** return the current time warp factor */
double GetMOOSTimeWarp();

/**pause for nMS milliseconds */
void MOOSPause(int nMS,bool bApplyTimeWarping = true);

/**return time as a double (time since unix in seconds). This will
 also apply a skew to this time so that all processes connected to a
     MOOSCommsServer (often in the
 shap of a DB) will have a unified time. Of course if your process isn't using
     MOOSComms
 at all this funtion works just fine and returns the unadulterated time as you
     would expect**/
double MOOSTime(bool bApplyTimeWarping=true);

/** call this to disable or anble high precision windows timers. By default they
    are on and always used in
calls to MOOSTime()*/
bool SetWin32HighPrecisionTiming(bool bEnable);

/**return high precision timestamp - time since unix in seconds only has high
    precision in win32*/
double HPMOOSTime(bool bApplyTimeWarping = true);

/** Return time as a double (time since unix in seconds). This returns the time
 as reported by the local clock.  It will *not* return time at the Comms Server,
 as MOOSTime tries to do. **/
double MOOSLocalTime(bool bApplyTimeWarping=true);


/**useful keyboard trap*/
int      MOOSGetch();


////////////////////// OUTPUT TOOLS ////////////////////////////////
//formatted printing
/** print a string*/
void MOOSTrace(std::string  Str);

/** print a formatted string (with printf-like format codes) and to debug window
    in DevStudio*/
void MOOSTrace(const char *FmtStr,...);

/** return a formatted string (with printf-like format codes*/
std::string MOOSFormat(const char * FmtStr,...);

/** Inhibit (enable) MOOSTracing in the calling thread*/
void InhibitMOOSTraceInThisThread(bool bInhibit = true);

/** like MOOSTrace but returns false - useful for return statements */
bool MOOSFail(const char * FmtStr,...);

/** return nicely formatted time stamp string */
std::string MOOSGetTimeStampString();

/** get the current date formatted nicely */
std::string MOOSGetDate();

/** useful macro for debugging prints line and file */
```

```
#define  MOOSHERE  MOOSFormat("File␣%s␣Line␣%d", __FILE__,__LINE__).c_str()

/** print a progress bar  - dfPC is the percent of a job completed*/
void Progress(double dfPC);


//these are used to let people format string used to control
//actuation (via a Thirdparty task) - one has to question why they are here
    though..
std::string MOOSThirdPartyActuationString(double * pdfRudder,double * pdfElevator
    ,double * pdfThrust);
std::string MOOSThirdPartyStatusString(std::string sStatusCommand);


////////////////////// NUMERICAL TOOLS  ////////////////////////////////

/** Bound angle to +/-PI*/
double MOOS_ANGLE_WRAP(double dfAng);

/** convert deg to rad*/
double MOOSDeg2Rad(double dfDeg);

/** convert rad 2 deg */
double MOOSRad2Deg(double dfRad);

/** Bounds |dfVal| < dfLimit but keeps sign. Returns true if it was clamped */
bool MOOSAbsLimit(double & dfVal,double dfLimit);

/** returns sample fom Gaussian process strength Sigma mena zero*/
double MOOSWhiteNoise(double Sigma);

/** returns x  for probablity mass such p(v<=x) = dfArea)*/
double MOOSNormalInv(double dfArea);

/** generates uniform noise in integers between interval nMin->nMax */
int MOOSDiscreteUniform(int nMin, int nMax);

/** generates uniform noise in interval dfMin-dfMax */
double MOOSUniformRandom(double dfMin, double dfMax);

/** Clamps a templated type between two values */
template <class T>
const T& MOOSClamp(const T &val, const T &min, const T &max)
{
    if (val < min) return min;
    if (max < val) return max;
    else return val;
}


////////////////////// FILE SYSTEM TOOLS  ////////////////////////////////

/** fills in a string list of all regular files found in specfied path
if bFiles==true only files are returned, if bFiles = false only directories
are returned*/
bool GetDirectoryContents(const std::string & sPath,std::list<std::string> &
    sContents,bool bFiles= true);

/** make a directory */
bool MOOSCreateDirectory(const std::string & sDirectory);

/** splits a fully qualified path into parts -path, filestem and extension */
```

```
bool MOOSFileParts(std::string sFullPath, std::string & sPath,std::string &sFile,
    std::string & sExtension);



//////////////////// MISC TOOLS ////////////////////////////


/** templated function which swaps byte order of type T returning it*/
template <class  T > T SwapByteOrder(const T &v)
{
    T r = v;
    char * aR = (char*)&r;
    std::reverse(aR,aR+sizeof(T));
    return r;
}


/** returns true if current machine is little end in*/
bool IsLittleEndian();


/** Functor class for performing static_cast between two types.
    Use it with stl::transform when copying between two collections
    with different element types */
template<class D> struct static_caster
{
    template<class S> D operator()(S s) const { return static_cast<D> (s); }
};


/** Functor class for performing dynamic_cast between two types.
    Use it with stl::transform when copying between two collections
    with different element types */
template<class D> struct dynamic_caster
{
    template<class S> D operator()(S s) const { return dynamic_cast<D> (s); }
};


//adds a token/value pair to end of the supplied string sIn
template <class T>
std::string & MOOSAddValToString(std::string & sIn, const std::string & sTok,
    const T & Val)
{
    std::stringstream s;

    if(!sIn.empty())
                s<<",";

    s<<sTok<<"="<<Val;

    sIn+=s.str();

    return sIn;
}


///////// USEFUL MACRO FOR PREVENTING COMPILER WARNINGS
#ifdef _WIN32
#define UNUSED_PARAMETER(a) a
```

```
#else
#define UNUSED_PARAMETER(a)
#endif



#endif
```