Logging with pLogger

Paul Newman

September 28, 2007

1 Logging - pLogger

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

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

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

where *varname* is any MOOS variable name and *period* is the minimum interval between log entries that will be recorded for the given variable. For example if *varname*=INS_YAW and *period* = 0.2 then even if the variable is published at 20Hz it will only be recorded at 5Hz. The optional NOSYNC flag indicates that this variable should not be recorded in the synchronous logs (see section 1.2). The optional MONITOR flag tells pLogger to send a notification if this variable isn't logged at least every 10 seconds. The notification occurs under the MOOS_DEBUG variable. If you are running iRemote (which subscribes to this variable automatically) you'll see a warning printed to the screen. This can be pretty useful when runnning a complicated system and you really do want notification that an important variable isn't being logged (probably because the process producing the variable is kaput in some sense.)

1.1 Logging Session

The logger supports the notion of Logging Sessions. For each log session the Logger will create a new directory and place all the logged files within that directory. These are typically (see later sections) an alog file a slog file a system log file, a copy of the mission file (moos file) and if applicable a hoof file (if pHelm is running). A new log session can be created by writing the variable LOGGER_RESTART to the MOOSDB or if you are using iRemote by pressing shift-g.

1.2 Log File Types

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

1.2.1 Synchronous Log Files

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

`SyncLog` = *true/false @ period*

where *period* is the interval time.

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

1.2.2 Asynchronous Log Files

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

1. Records both string and numeric data
2. Records data in a list format - one notification per line
3. Entries only made when variable is written

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

1.3 System Log File

There is a third kind of log file that is produced - a system log file (ylog). This file only contains data contained in `MOOS_SYSTEM` and `MOOS_DEBUG` messages. The later can be written to by calling the `CMOOSApp::MOOSDebugWrite` method from any `CMOOSApp` derived class. The thinking behind the ylog files is that its pretty useful to be able to browse through a text file of events to see when and if things went wrong in a mission. Processes can write to `MOOS_SYSTEM` and or `MOOS_DEBUG` if something happens which they think is salient. Think of this as `/var/log` .

1.4 Dynamic Logging Configuration

Post V7.0.1 releases of MOOS include a dynamic logging ability. External parties can, at run time request the logger to start logging particular (named) variables. The logger subscribes to messages called `PLOGGER_CMD` (if the Logger is being run under the MOOS name “pLogger” otherwise subsitute the relevant name) and by correctly formatting the string data of this message dynamic logging can be invoked.

$$\texttt{LOG\_REQUEST} = \textit{varname} \ @ \ \textit{period} \ [NOSYNC], [MONITOR]$$

Note that this string format is identical to that found in the mission file modulo `LOG` being replaced with `LOG_REQUEST`. Now dynamic logging sits naturally with “alog” (asynchronous) files but not so well with “slog” files. These files are easy to use(and hence popular) because they are rectangular numerical array written to text file (easily parsed by Matlab’s load command). But dynamic logging means that by the time a mission ends an unknown number of variables (columns) will occur in every line of the slog. To address this issue the logger can be configured (in its mission file configuration block) with a line like

$$\texttt{DynamicSyncLogColumns} = 30$$

Here 30 columns are being reserved for variables that are requested at run time. As dynamic requests are received and processed by the logger these unclaimed columns are consumed until none remain. At this point any future dynamic logging requests will still be accepted but the logged variables won’t appear in the slog file (they will of course appear in the alog file). Unclaimed slots will be labelled `DYNAMIC_X` until claimed and will always have `NaN` entries.

1.5 Specifying Log File Names and Locations

Each time the logger starts creates (if required) a new directory in the logging root directory(see below) and performs logging within that directory. `pLogger` is quite flexible in terms of log file configuration and is controlled by the following variables.

`GlobalLogPath` This is a **file scope variable** (ie not in any process configuration block). If it is present in a mission file then it specifies the root directory in which log files will be created.

`Path` specifies the root logging directory but only used if `GlobalLogPath` is not set (see above).

`File` this is the stem file name given to logged files (alog/slog and ylog)

`FileTimeStamp` if this is set to true, the name of each logged file and (created containing directory) will be the concatenation of the `File` variable and a time stamp. If `FileTimeStamp` is false then each time the logger is run it will write to the same set of log files **and destroy the original contents**. This is by design, when developing it’s often useful to not have useless log files take ever more space on you machine.

Note that if the logger is run without a mission file it starts logging to the local directory with file time stamping enabled. See figure 1.7 for a typical logger configuration block.

If for some reason teh logger was unable to start logging to a location specified in the log file it trys, as a last resort, to open a log directory in the current working directory ('.'). If this fallback fails then all is lost, we are without hope and the Logger exits. If you are running a mission from pAntler you'llbe notified that the logger has quit.

1.6 Mission Backup

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

1.7 Example Configuration

2 Runtime File Backup

In Post 7.0.1 releases, by writing a correctly formatted string value to the MOOS variable `PLOGGER_CMD` processes withing the MOOS community can request the logger to back up arbitrary files to the current log directory. For example if the contents of a message with name `PLOGGER_CMD` is set to

```
COPY_FILE_REQUEST = /home/pnewman/code/TheFile.xzy
```

and published to the `MOOSDB` it will result in a copy of '/home/pnewman/code/TheFile.xzy' being placed in the logging directory under the name 'TheFile._xzy'- note the additional underscore. If the file in question has no extension then '._bak' is added appended to the file name.

3 Replay – `uPB`

There is a FLTK-based, cross platform GUI application that can load in *alog* files and replay them into a MOOS community as though the originators of the data were really running and issuing notifications. A typical use of this application is to "fake" the presence of sensor processes when reprocessing sensor data and tuning navigation filters. Alternatively it can be used in pure replay mode perhaps to render a movie of the recorded mission. The GUI allows the selection of which processes are "faked". Only data recorded from those applications will be replayed from the log files. There is a single class that encapsulates all the replay functionality - `CMOOSPlayback`. The GUI simply hooks into the methods exported by this class. The GUI is almost self documenting - start it up and hold the mouse over various buttons.

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

```
//////////////////////////////////////////////
// Logger configuration block
ProcessConfig = pLogger
{
    //over loading basic params..lets be fiesty
    AppTick  = 20.0
    CommsTick   = 20.0

    //all file names begin with this stem
    File        = SciPark29Mar

    //where is the root log directory
    PATH        = /home/doe/MOOSData/SciencePark/

    //yes we want some sync logging for crude
    //performance checking
    SyncLog      = true @ 0.2

    //yes we want async logging so we can replay
    // exactly what happened and record strings
    AsyncLog     = true

    //yes append each created directory log file
    //with a time stamp DAY_MONTH_YEAR_TIME
    FileTimeStamp    = true

    //what do we want to log
    //(zero means capture everything)
    Log       = LMS_LASER_2D @ 0 MONITOR
    Log       = LMS_LASER_3D @ 0 MONITOR
    Log       = MARGE_ODOMETRY @ 0
    Log       = DESIRED_RUDDER @ 0
    Log       = DESIRED_THRUST @ 0
    Log       = CAMERA_GRAB @ 0
    Log       = GPSData @ 0.4

}
```

Figure 1: A Typical pLogger configuration block

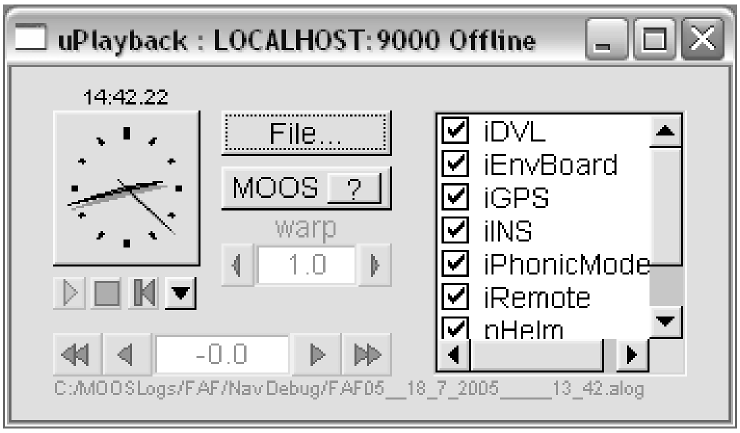

Figure 2: A screen shot of `uPB` - a cross platform “alog” playback tool