Makani Telemetry Users' Guide

The Makani Simulator and Flight Controller produce telemetry which can be interpreted in real-time using the visualizer and flight monitors, and stored for later analysis in log files in the hdf5 format. The telemetry log files are the primary vehicle for analyzing and debugging the system. While most signals of interest are already available in the telemetry, adding new signals is easy to do if required. The flight controller logs produced in simulation and in real flight operation are identical in format.

This document serves as a quick introduction and reference to some of the more commonly used telemetry signals. After running a simulation, this guide will help you quickly locate and plot signals of interest.

Sources of Code and Data

After the shutdown of Makani, we have released a final snapshot of the complete software repository and a representative collection of flight data log files. These may be obtained from the following URLs.

Makani public software release	**https://github.com/google/makani**
Makani public flight data release	**https://console.cloud.google.com/marketplace/details/makani-flight-logs**

Introduction to the Telemetry Structure

All of the avionics nodes in the system send asynchronous messages in the AIO format over the network.[1] Makani log files are constructed by first simply recording the network traffic using tcpdump forming a packet capture (pcap) file. This pcap file is then converted to the format hdf5, which presents an interface with aspects both of a hierarchical tree and of a time series. These data files can be easily inspected with MATLAB, Python, or in other environments. In Python the log file data presents a dictionary-like interface; in MATLAB, a struct. The use of tab-completion for exploring the structures is all-but-necessary.[2]

When running a simulation, the simulator and the flight controller also communicate using AIO messages. In a simulation, messages of type SimSensorsMessage contain ersatz sensor data sent from the simulator to the flight controller. The simulator also broadcasts the true values of

[1] The avionics network and message format is described in "A Low-Cost Fiber Optic Avionics Network for Control of an Energy Kite" by Kurt Hallamasek *et al*, also included in this volume.
[2] See `lib/python/ipython_completer.py` for tab-completion of HDF5 fields in Python. This add-on is a must-have.

the simulator state in the message SimulatorTelemetry. The flight controller sends its diagnostics in a message called ControlTelemetry, which is identical in format whether a flight is real or simulated.

For the purposes of flight analysis, most telemetry of interest is found in the ControlTelemetry sent by the flight controller, running on node ControllerA. This guide will focus on this control telemetry.

The ultimate source of truth regarding these data structures is the code itself. Indeed, I'm writing this document by referencing the source code; the intent is for the code to be self-documenting. Instructions for obtaining the code are included at the end of this document.

One small caution: when loaded in MATLAB, array structures in the log files tend to be transposed as compared to how they are loaded in Python, or how the data structures are represented in C. The major user-visible consequence of this is that all matrices loaded in MATLAB from the h5 file will be transposed.

message type	*file containing definition(s)*
Control Telemetry	control/control_telemetry.h
Simulator Telemetry	sim/sim_telemetry.h
Avionics messages (approximately 134 types)	avionics/common/avionics_messages.h

Structure of the Log Files

The log files are structured as a tree. Here we dive down into the tree to get the control telemetry, and in the process describe the general structure of the log file.

1. At the top level you'll find three fields:

- **bad_packets**. Data that could not be parsed.
- **messages**. The actual telemetry messages.
- **parameters**. All configured parameters, such as the wing mass, tether length, etc.

2. To get the telemetry, select **messages**. At the second level you will now find a list of nodes on the wing, such as **kAioNodeMotorPbi** (the port-bottom-inner motor controller, known as "PBI"), **kAioNodeServoE1** (one of the two servos driving the elevator), and **kAioNodeControllerA** (the primary flight controller).

3. Now select one of these nodes. Here we'll select **kAioNodeControllerA**, which runs the flight controller. Now, at the third level, you will now see a list of all the message types sent by the flight controller:

kMessageTypeControlSlowTelemetry kMessageTypeControllerCommand
kMessageTypeControlTelemetry kMessageTypeQ7SlowStatus
kMessageTypeControlDebug

Of these, **kMessageTypeControlTelemetry** and **kMessageTypeControlDebug** represent the control telemetry, at 10 Hz and at 100 Hz, respectively. If you're examining flight data, the higher rate data is only available from the wing recorder log, as the high-rate data is not sent over the radio link from kite to ground during a flight. These two message types are identical except for the rate at which they are sent. Select one of them.

4. Now at the fourth level, there are three fields:

- **capture header**. Contains the message timestamps in seconds plus microseconds since some epoch a long time ago (Jan 1, 1970 UTC), and the destination and source IP addresses. An array of: tv_sec, tv_usec, source_address, destination_address. This timestamp is made by the computer logging the data (a laptop running Debian Stretch Linux). The latency from transmission to logging should be sub-100ms and the absolute timestamp is probably good to better than 1s.
- **aio_header**. Makani's AIO ("avionics I/O") message header. An array of: version, source, type, sequence, timestamp.
- **message**. This is the actual telemetry message. A structured array.

5. Finally, select "**message**". Now you have the actual control telemetry messages, whose structure is described below.

Structure of the Control Telemetry

The Control Telemetry (once again, defined in control/control_telemetry.h, which should be your go-to reference) contains several things:

1. **control_input** All input values used by the controller, converted to real units and useful coordinate systems. This includes the air data (pitot-static), rotor speeds, flap positions, wind, ground station encoder values, etc.
2. **control output**: All output values from the controller. Rotor speed commands and commanded flap positions.
3. **state_est**: The estimated state of the wing, including position, attitude, velocity, angle of attack, sideslip, etc.
4. **flight_mode**. The current flight mode. (See table of values below.)

5. Telemetry from the various flight controllers (hover, trans-in, crosswind, off-tether), and from the inner workings of the estimator.

Coordinate Systems, Units, and Naming Conventions

The two main coordinate systems you will encounter are (described here **for reference only**, see the link to repo below for canonical source):

- **Body coordinates**, indicated with a _b suffix on telemetry field names. X is forward, Y is towards starboard, and Z is down.
- **Ground coordinates**, indicated with a _g suffix, where for RPX flights at China Lake, X points approximately west (269°) and Z is down. Here the minus x axis is aligned with the GS-to-kite radial. The origin is on the tower axis of symmetry, "at the base of the slewing bearing." For the Parker Ranch (Hawaii) and Offshore test sites, the ground coordinate system is NED.

Further details of these coordinate systems and others in use are documented in the file control/coordinate_systems.md.

Units

All angles are in radians. All quantities in the controller telemetry are in SI units (meters, meters/second, Pascals, radians, radians/sec, etc).

Filtered Values

Low-pass-filtered values are indicated with an _f suffix.

State Estimate

The estimated state of the system is stored in the **state_est** structure, which is of type StateEstimate, defined in control/estimator/estimator_types.h.

Some fields of interest are:

field	*units*	*description*
Xg	vector of meters	Wing **position**, in ground coordinates.
Vg	vector of m/s	Wing **velocity**, in ground coordinates, i.e. an estimate of the time derivative of Xg.
Vb	vector of m/s	Wing velocity, in body coordinates, i.e. dcm_g2b · Vg.

Vb_f	vector of m/s	Wing velocity, in body coordinates, low-pass-filtered. The filter is defined in config/m600/control/estimator.py (see Vb_filter_a and Vb_filter_b) and currently consists of a 2nd order butterworth low-pass at 8.3 Hz.
Ag	vector of m/s²	Wing **acceleration**, in ground coordinates
Ab_f	vector of m/s²	Wing acceleration, in body coordinates, low-pass-filtered.
dcm_g2b	3×3 matrix	**Attitude**, expressed as the 3×3 rotation matrix that rotates ground coordinates into body coordinates. Use the Matlab function "dcm2angle" (from the Aerospace Toolbox) to convert this to roll/pitch/yaw Euler angles if desired. Remember that matrices will be transposed when loaded from h5 into MATLAB.
pqr	array of rad/s	Angular rates. P, q, and r are the rotation rates around the body x, y, and z axes (respectively), in rad/s, using the right-hand-rule.
tether_force_b	structure	Estimated force of the tether on the wing, in body coordinates. This structure has these fields: tension [N] - the total tension roll [radians] pitch [radians]
apparent_wind	structure	Structure containing the apparent wind estimate, both as a velocity vector in the body frame [m/s], and also in spherical coordinates (airspeed, alpha, and beta). The pitot probe dynamic pressure measurements are converted to airspeed using a hard-coded value of the mean density at this particular test site. Thus it is something between indicated airspeed and true airspeed. The intent is that apparent_wind be treated as a true airspeed directly comparable to the velocity measurements obtained from GPS and the inertial navigation system.
stacking_state		Indicates whether a motor has faulted and been taken out of the "stack."
acc_norm_f		Magnitude of the wing acceleration, low-pass filtered.
joystick		Joystick stick and switch positions.
perch_azi	rad	Direction the perch is pointed.

winch	structure	State of the winch
wind_g	structure	Vector, speed, and direction of wind, in ground coordinates, observed at the wind sensor on the ground station.
vessel	structure	State of the buoy (offshore only), including its position, velocity, attitude, and angular velocity. These states define the relationship of the vessel frame with respect to the ground frame.

Note on Loop Angle:

0 degrees = 9 o'clock = where we trans-in
90 degrees = 6 o'clock = bottom of the loop
→ Loop angle decreases as we fly forward.

Control Input

The minimally-processed (but not completely raw) input to the controller is reported in the control_input field (of type ControlInput, defined in control/control_types.h).

field	*type or units*	*description*
imus	structure	Inertial measurement unit data, in body coordinates. There are three independent IMUs. The fields of this structure are: acc: acceleration vector (m/s^2, body coords) gyro: angular velocities mag: magnetic field measurement (Gauss)
pitots	structure of Pascals	Air data system. There is only one air data probe (aka Pitot tube), but the ports are plumbed to two separate sets of sensors: one high-sensitivity, low-range; the other lower sensitivity but higher range. All values are in Pascals. NOTE. In 2017, three auxiliary Pitot tubes were installed (on each wingtip, and on the horizontal stabilizer). Data from these sensors is **not** present here. These sensors are completely ignored by the controller.
flaps	array of radians	Current flap positions. The order is [A1, A2, A4, A5, A7,

		A8, Elevator, Rudder], where A1 is the port wingtip aileron, and A8 is the starboard wingtip aileron. A3 and A6 are not reported as their positions are fixed. Positive numbers indicate trailing edge down (ailerons and elevator) or to port (rudder). The gear reduction in the rudder ***is*** accounted for here; this is the position of the **control surface**.
rotors	array of radians/sec	Measured rotor speeds. Not used by the controller but useful for offline analysis. The order of motors is: [SBO, SBI, PBI, PBO, PTO PTI, STI, STO], where the nomenclature denotes Starboard/Port, Inner/Outer, Top/Bottom.
wind_ws	vector of meters/sec	Wind velocity vector measured by the weather station, in its own coordinate system. See state_est.wind_g for the wind as converted to ground coordinates.
loadcells	array of Newtons	Raw forces seen by the various loadcell axes at the bridle points. See state_est.tether_force_b to get the total tether force and direction.
gsg	structure	Azimuth and elevation (in radians) of the ground-side-gimbal (GSG). Azimuth increases when rotated with the right-hand-rule about the down (+z) axis and has an arbitrary zero position; elevation increases when moved upwards and is zero when horizontal.
wing_gps	structure	Solution information from the two GPS receivers onboard the wing (position [m] in ECEF, velocity [m/s, ECEF], sigmas, and solution type). If you're interested in the wing position, however, you should use state_est.Xg [m, ground frame].
gs_gps	vector	Ground station GPS location [m] ECEF.
tether_released	bool	True when the tether has been released.
joystick	structure	Positions of joystick sticks and switches.
perch	structure	Perch data. The only value here currently relevant is perch_azi, giving the values of the perch azimuth encoders. Other fields pertain to GSGv1 hardware: perch heading (if there's a perch compass), proximity flag, winch position, and levelwind elevation.

sync		

Control Output

The ContolOutput structure contains the controller's output commanding the various actuators (servos, motors, winch, and detwist drive). For the measured positions and speeds of these actuators see control_input; the corresponding structures may be compared to check command following of the motors, servos, winch, etc.

field	*units*	*description*
flaps	radians	Array of commanded positions for all the flaps. Order is the same as in control_input.
rotors	radians/sec	Array of commanded motor speeds. Order is same as in control_input. Sign convention is:
winch_vel_cmd		Commanded winch velocity. Not used with tophat.
detwist_cmd	radians	Commanded tether detwist angle.
run_actuators	bool	
tether_release	bool	Whether to command tether release.
light	bool	Whether to turn on the FAA visibility light. Not used.
sync		Not used?

Crosswind telemetry

ele_tuner		Not used
airspeed_target		
path_type		
path_center_g		
mean_airspeed_cmd	m/s	
loop_angle	radians	
eulers_cw		
target_pos_cw		
current_pos_cw		
k_geom_cmd		
k_aero_cmd		
k_geom_curr		
k_aero_curr		
pqr_cmd	radians/sec	
alpha_cmd	radians	
beta_cmd	radians	
tether_roll_cmd	radians	
airspeed_cmd	m/s	
thrust_ff	Newtons	
thrust_fb	Newtons	
int_elevator		
int_rudder		
int_aileron		

int_thrust		
deltas		

RPX-04 Experimental Test Configurations

For RPX-04 and forward, we implemented a system allowing us to choose "experimental test configurations" from an enumerated set of possibilities. Each test configuration changes the nominal angle-of-attack (alpha) and/or airspeed commands. **Without access to previous revisions of the Makani code repository, it is generally impossible to decode these test cases, as they changed from flight-to-fight**.

How to find out what test configuration was staged

Look at ControlTelemetry → state_est → experimental_crosswind_config [#]. This is a number indicating which configuration number was staged.

How to find out whether it was active

Look at ControlTelemetry → state_est -> joystick -> pitch_f < -0.5

In MATLAB to get active test configuration index:

```
int32(control_debug.message.state_est.experimental_crosswind_config) .* ...
    int32(control_debug.message.state_est.joystick.pitch_f < -0.5)
```

See Also

Take a look at the RPX-04 Flight Controls Test Plan.

Table of Flight Modes

Unfortunately, enumerations are not stored in the h5 log files; you generally have to reference the source code to figure out what servo index corresponds to which servo, or which number corresponds to which flight mode. This is a long-standing issue that was never addressed. In particular, because the mapping between integer values and their logical interpretations changed from time to time, one must check out the particular version of code used for a particular flight in order to be sure of the meaning.

Here's a table of the current mapping of flight modes, but this may change in the future. The definitive source of truth is FlightMode in control/control_types.h.

flight_mode	*description*
0	Pilot Hover
1	Perched
2	Hover Ascend
3	Hover PayOut
4	Hover Full Length
5	Hover Accel
6	Trans-In
7	Crosswind Normal
8	Crosswind Prep-Trans-Out
9	Hover Trans-Out
10	Hover Reel-In
11	Hover Descend
12	Off Tether
13	Hover Transform Gs Up
14	Hover Transform Gs Down
15	Hover Prep Transform Gs Up
16	Hover Prep Transform Gs Down

Parameters

Parameters are also recorded in the log file. Arturo asks:

> *Hey Tobin,*
>
> *Are kite inertias, mass, CG location, etc part of the data stream, or are they just internal constants in the code? Either way, how can I find them?*
>
> *thanks!*

These numbers are set in the configuration system (in the "config/" subdirectory) and recorded in the "parameters" structure of the log.

Take a look at config/m600/wing.py to see the configured values. You'll find:

```
  # The wing_mass and center_of_mass_pos estimates are based on the mass
# tracker [internal ref] on 2016-11-14. The configuration does not include
# any mass balance but includes the landing gear.

# Wing mass [kg] includes both the rigid wing (1579.8 kg) and
# tether release and bridle (50.4 kg).
wing_mass = 1579.8 + 50.4

# Center-of-mass [m] for the low-tail configuration.
center_of_mass_pos = [-0.154, 0.009, 0.092]

# These values are from 016-09-11 and use the
# ASWING model M600_r09-7_xwind.asw
reference_wing_mass = 1560.7
I = np.array([[30140.0, 5.0, 28.3],
              [5.0, 9044.0, 27.1],
              [28.3, 27.1, 36510.0]]) * wing_mass / reference_wing_mass
```

These values are also recorded in the log file:

```
In [1]: import h5py

In [2]: log = h5py.File('20161121-142912-flight01_crosswind.h5', 'r')
```

```
In [4]: log['parameters']['system_params']['wing']['I']['d']
Out[4]:
array([[[  3.14821734e+04,   5.22265650e+00,   2.95602358e+01],
        [  5.22265650e+00,   9.44674108e+03,   2.83067982e+01],
        [  2.95602358e+01,   2.83067982e+01,   3.81358378e+04]]])

In [5]: log['parameters']['system_params']['wing']['center_of_mass_pos']
Out[5]:
array([(-0.154, 0.009, 0.092)],
      dtype=[('x', '<f8'), ('y', '<f8'), ('z', '<f8')])
```

Practicalities

How to Look at a Log File Using …

Python

Here's a small example showing how to load an h5 log file in Python and plot a variable (in this case, the kite's altitude). Accessing the log files is made much less painful by enabling tab-completion of telemetry fields; instructions are in lib/python/ipython_completer.py.

```
import h5py
import pylab

log = h5py.File('20161121-142912-flight01_crosswind.h5', 'r')
C = (log['messages']['kAioNodeControllerA']
       ['kMessageTypeControlTelemetry']['message'])
pylab.plot(C['time'], -C['state_est']['Xg']['z'])
pylab.show()
```

Additionally, by starting Python with "bazel-bin/lib/bazel/pyembed ipython", you will be able to access some Makani library functions (like DcmToAngle) directly from Python. Ask in the controls/avionics offices for more info.

You can also explore the field names using .items() or .keys() and .dtype as appropriate:

```
log.keys()  # Shows [u'bad_packets', u'messages', u'parameters']
log['messages'].keys()  # Shows [u'kAioNodeBattA', … ]
```

```
# The number of messages of this type in the h5 file:
log['messages/kAioNodeBattA/kMessageTypeSlowStatus'].len()

# The first message:
log['messages/kAioNodeBattA/kMessageTypeSlowStatus'][0]

# The nested field names:
log['messages/kAioNodeBattA/kMessageTypeSlowStatus'][0].dtype
```

MATLAB

Here's a small example showing 3 ways to load an h5 log file in MATLAB and plot a variable (in this case, the kite's altitude).

Method 1 (quick to load a specific telemetry dataset to workspace with MATLAB built in function):

```
C = h5read('20161121-142912-flight01_crosswind.h5',
'/messages/kAioNodeControllerA/kMessageTypeControlTelemetry');
time = C.message.time;
altitude = -C.message.state_est.Xg.z;
figure;
plot(time, altitude)
```

Method 2 (takes long to load the data to workspace but *all* telemetry data is accessible on load):

- NOTE: Only works in MATLAB 2016a and earlier. Find out your MATLAB version by running "ver" on the console. See bug 31990348 if you have spare cycles to help fix this.
- You need the makani repository loaded to your computer (see 'How to get the code' section below).
- Open MATLAB and navigate to the following directory on the console:
  - $MAKANI_HOME/analysis
- Run the following script in the MATLAB console to set all relevant paths:
  - SetMatlab
- Now you can run the following code on console to access telemetry data:

```
log = h5load('20161121-142912-flight01_crosswind.h5');
C = log('/messages/kAioNodeControllerA/kMessageTypeControlTelemetry');
Time = C.message.time;
Altitude = -C.message.state_est.Xg.z
figure;
plot(Time, Altitude)
```

Method 3 (the best of both worlds! Lazily loads *all* datasets *quickly*):

- You need the makani repository loaded to your computer (see 'How to get the code' section below).
- Open MATLAB and navigate to the following directory on the console:
    - $MAKANI_HOME/analysis
- Run the following script in the MATLAB console to set all relevant paths:
    - SetMatlab
- Now open the H5Plotter (a GUI interface for opening and plotting H5 log data) by running the following in the MATLAB console:
    - H5Plotter
- Load a H5 log file using the 'Choose' button in the top right corner.
- Once the file is loaded, datasets appear in 'AIO Nodes' panel box. Click on one or more of these nodes to access the corresponding datasets in the 'AIO Messages' panel box. Only datasets common to all selected AIO nodes are shown.
- Once you have selected data to plot in the 'AIO Messages' panel box, use the 'plot' button at the bottom right corner to visualize the data. Holding ctrl or shift allows multiple fields to be selected and selecting a node in the tree will plot all data contained beneath that node. Data can also be exported by right clicking.
- NOTE: You can plot multiple datasets simultaneously on the same time axes. How many datasets you can plot at the same time is only limited by your machine's RAM; be judicious about this.

Matlab Example: Plot roll, pitch, and yaw.

Here's a small example that converts the dcm_g2b matrix into Euler angles.

```
% Read the log file.
filename = '20161121-142912-flight01_crosswind.h5';
C = h5read(filename, ...
   '/messages/kAioNodeControllerA/kMessageTypeControlTelemetry');

% Fetch the dcm_g2b matrix
dcm_g2b = C.message.state_est.dcm_g2b.d;

% Transpose the dcm_g2b matrix.
dcm_g2b = permute(dcm_g2b, [2 1 3]);

% Compute Euler angles.
[yaw, pitch, roll] = dcm2angle(dcm_g2b, 'ZYX');

% Plot the results.
plot(C.message.time, roll  * 180/pi, '.', ...
    C.message.time, pitch * 180/pi, '.',...
    C.message.time, yaw   * 180/pi, '.')
```

```
legend('roll', 'pitch', 'yaw')

ylim([-180 180]);
set(gca, 'YTick', -180:30:180);
grid on;
xlabel('controller time [s]');
ylabel('angle [degrees]');
title(['flight attitude (' filename ')'], 'interpreter', 'none');
```

Auxiliary Sensors

After rpx-02 but prior to rpx-03, three additional sensor groups were attached to the wing: one on each wing tip and one on the horizontal stabilizer (elevator?). These were removed before CW-01. These sensors include:

Port wingtip	5-port air data probe, hover orientation GPS IMU and magnetometer
Starboard wingtip	5-port air data probe, crosswind orientation GPS IMU and magnetometer
Tail	5-port air data probe (* some channels not plumbed)

Other Documentation

ECR 170	Engineering change request for installation of **wingtip pitots** + other stuff.
ECR 177	Engineering change request for installation of **elevator pitot** + other stuff.

Example Uses of These Sensors

- Determine the actual wind speed at the kite during hover.
- Additional airspeed measurement during crosswind less affected by props/main wing?
- Use wingtip GPS's to partially determine kite attitude.

Where to Find the Data

Data from these auxiliary sensors is currently ignored by the flight controller (including the state estimator); you have to look at the raw avionics telemetry from the relevant nodes. These sensors were implemented by piggybacking on the kite wingtip and tail lighting nodes. The node names are **LightPort**, **LightStbd**, and **LightTail**. The structure of the messages is similar to those from the main flight computers (FcA, FcB, FcC) which host the main GPS receivers, IMUs, and nose cone air data system.

Known Issues

rpx-03	[bug 35893018] Wingtip GPS solution quality is poor due to self-jamming by the avionics. [bug 37252007] Plumbing issue in the tail pitot tube?

Sensor Metadata

Starboard pitot tube (front-facing):
Position: (302, 12801, -393)
Orientation: Pitched 3 degrees down from kite +x axis (same as pitot on massbalance tube)

Port pitot tube (down/hover-facing):
Position: (108, -12802, -221)
Orientation: Pitched 3 degrees backwards from kite +z axis (90 deg from starboard pitot)

GPSs:
Positions: (0, +/-12809, -555)
Orientation: Parallel to -z axis

Starboard IMU
Position: (-95, 12805, -447)
Orientation: I don't know which way is front/back/left/right of the IMU. But relative to the PCB sitting on a table, it is rolled 94.5 degrees to the left (as in port wingtip down, starboard wingtip up).

Port IMU
Position: (-95, -12805, -447)
Orientation:Relative to the PCB sitting on a table, it is rolled 94.5 degrees to the right (as in port wingtip up, starboard wingtip down).

Timebases

For analysis of rpx-01 data, many groups have used different timebases for presenting flight data. We should standardize the timebase for analysis for rpx-02. Some options:

Method 1: Time since start of log file

This is what the web log viewer uses, and consequently became the de facto standard for rpx-01 analysis.

Pros: Easy.

Cons: Inconsistent across multiple log sources (command center, wing, platform recorder); arbitrary; depends on which messages you're looking at.

Method 2: Actual wall-clock time as determined by recorder computer.

Every logger timestamps every packet with the capture time, as determined by that particular logger computer. This depends on both the actual time the packet was received at that logger, and the local clock on that logger machine.

How to: The capture_header structure associated with each message includes two integer values giving the capture timestamp: **tv_sec** and **tv_usec**, which together give the time in seconds and microseconds since the Unix epoch, January 1st 1970. Combine them using **t = tv_sec + 1e-6 * tv_usec** to get a timestamp in seconds and fractional seconds since that epoch.

Pros: Reasonably accurate and syncable to other sources.

Cons: Can be offset by several seconds depending on accuracy of local clock.

Method 3: Actual wall-clock time as determined by GPS

The timestamps from Method 2 can be cross-checked with GPS timing information included in the log file.

Method 4: Controller Time

End of document.