

<b>Introduction</b>	<b>1</b>
<b>Unstructured Log API</b>	<b>2</b>
<b>Structured Log API</b>	<b>2</b>
Example Code	3
<b>OpenPilotReview</b>	<b>4</b>
<b>Further Work</b>	<b>4</b>

## Introduction

Logging in this context refers to an API that allows a C/C++, Python program to generate a message that will eventually be stored in a log file. The message may be an ascii string or a binary serialized object. The logging infrastructure will run independently on both the operator station (Windows) and vehicle (Linux), but will use common protobuf definitions and logging format. This allows use of a single set of tools. OpenPilot logging on a comma device will remain separate to avoid a requirement to track changes in a merged system.

Messages for the end user like help messages, error messages are not considered part of logging api and need to be handled separately.

These logging api(s) are implemented in a separate object called the “Logger(s)”. The various programs (called “clients” henceforth) are provided to handle these logger instances. The logger handles the following functionality.

- Verbosity level functionality. This could be set at command line or environment variable and will cause certain messages to be suppressed.
- Different standard formats like csv, json etc depending on whether the end user is a program or human.
- Thread safety. Multiple threads/processes can share the instance to the logger and so output should not get garbled.
- Maintenance of log files like rotation etc.
- Portability. Handle differences between different env like Win32, Linux.

Depending on the end user, there are 2 types of logging api.

- Unstructured Log API
- Structured Log API

# Unstructured Log API

This “printf” style api allows clients to write messages that are meant for consumption by users that have access to the source code of the program. The log files are typically used to debug systems by members of the internal team.

The actual message is

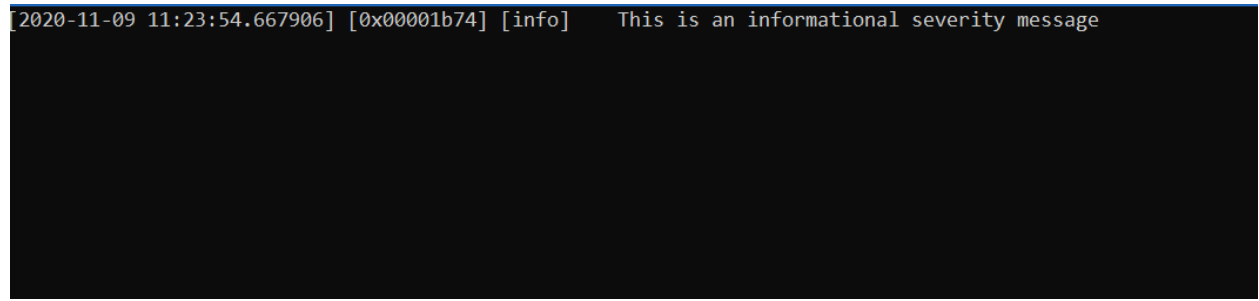
- unstructured string that has meaning to programmers.
  - It is recommended that programmers follow a consistent format at least within the same module, but **this is not enforced** by api.
- Same level of difficulty as a printf.
- Output log file is ascii.

For C/C++ programs the popular choice for this is [Boost.Log](#) This provides all the above functionality of a logger and has an easy to use api. For python we can use the standard [Logging](#) module.

For example a programmer can just use this trivial api to write out messages.

```
BOOST_LOG_TRIVIAL(trace) << "A trace severity message";  
BOOST_LOG_TRIVIAL(debug) << "A debug severity message";  
BOOST_LOG_TRIVIAL(info) << "An informational severity message";  
BOOST_LOG_TRIVIAL(warning) << "A warning severity message";  
BOOST_LOG_TRIVIAL(error) << "An error severity message";  
BOOST_LOG_TRIVIAL(fatal) << "A fatal severity message";
```

This will produce the following output in a standard format with the timestamp and thread id.



```
[2020-11-09 11:23:54.667906] [0x00001b74] [info] This is an informational severity message
```

# Structured Log API

This log file is for consumption by external developers or other post processing programs. The developers generally have a high level understanding of the architecture but may not have access to the source code of the program. The post processing programs are generally used for collecting statistics, training data, and debugging problems involving multiple processes by providing an overall view of the system etc.

The actual message is

- Structured with static data types..
  - Consistent schema **enforced** by api.
  - Versioning supported.
- Need to be declared in a central place to allow it to be shared.
- Needs multiple lines of code to send the message.
- Output log file is binary and needs to be decoded into ascii/json standard formats.

The popular choice for such is [capnproto](https://capnproto.org/cxx.html). This package is designed for distributed application and so includes both

- Serialization Layer
  - Message schema description language with static data types, lists, structs, enums.
  - Tool generates the various set/get routines in the language of choice.
- Communication.
  - Cap'nProto RPC layer implementing the [RPC Protocol](https://capnproto.org/rpc.html)

For logging purposes we are only interested in the serialization layer. Instead of using the RPC protocol we simply send the message to a file descriptor.

## Example Code

Below is an example code from <https://capnproto.org/cxx.html> where we build a Person message with fields like name, email, phones.

```
::capnp::MallocMessageBuilder message; // Create a message.

Person::Builder bob = people[1];      // Example person object
bob.setId(456);
bob.setName("Bob");
bob.setEmail("bob@example.com");
auto bobPhones = bob.initPhones(2);
bobPhones[0].setNumber("555-4567");
bobPhones[0].setType(Person::PhoneNumber::Type::HOME);
bobPhones[1].setNumber("555-7654");
bobPhones[1].setType(Person::PhoneNumber::Type::WORK);
bob.getEmployment().setUnemployed();

writePackedMessageToFd(fd, message);  // Writes to file descriptor
```

The binary log file can then be decoded by post processing scripts using this API

```
::capnp::PackedFdMessageReader message(fd); // Read from file descriptor
```

# OpenPilot Review

[OpenPilot](#) is an open source software for autonomous vehicles. It is an integral part of the remnav software stack, and programmers would need to understand its event reporting for debug purposes.

OpenPilot uses capnp to log events in the system in [log.capnp](#). It defines one struct called Event which will have the log time and a valid flag. Then a union defines the packet type.

It then uses a single publisher and multiple subscriber messaging implemented using ZMQ and their custom msgq. Logging the events is done by creating a loggerd that subscribes to all the ZMQ message sockets and writes them out to the logs.

We can follow the similar style of event reporting in our structured logging api.

## Further Work

In case the clients are on different machines, the file descriptor can be replaced by a windows or linux socket file descriptor. Pros and cons of using either TCP or UDP needs to be evaluated. But that is beyond the scope of this document.