# PVLogger User's Guide
# OHA 1.0, rev. 1
# Oct 20, 2008

# Table of Contents

# List of Figures

# 1. Introduction

This document describes the PVLogger functionality within OpenCore for logging and instrumentation. The key features of the design include:

- a very flexible hierarchical logging control,
- runtime control over logging levels,
- multiple logging destinations,
- hooks for message filtering,
- and control over message formatting.

The motivation behind building this capability into a low-level library within OpenCore, within the OS compatibility layer (OSCL), is to provide the common logging capability at the lowest layer possible in the code so it is as widely available as possible.

# 2. Overview

Within embedded software environments, such as those found on handsets and other portable devices, the debugging tools are often limited especially for devices near their final commercial configurations. While logging tools and instrumentation are always helpful, it is very important in these situations to have a convenient way to get information on what is happening in the system while minimizing the impact on performance. It also very useful to have the ability to log to multiple destinations such as memory buffers, files, serial connections, etc.

PVLogger has been designed to create a very flexible scheme for logging and instrumentation that can tradeoff performance impacts with the level of information provided. It allows the information to be routed to a variety of locations, log to multiple locations in the same run, etc. Another important feature is the ability to flexibly focus the attention of the logging in very targeted areas of the code to avoid information overload and minimize performance impacts. PVLogger has many similarities to opensource logging libraries such as log4j and log4cpp.

The key features provided by PVLogger include the hierarchical logging control, the ability to log to multiple destinations, the ability to flexibly format messages, and the ability to flexibly filter messages to any logging destination. In addition to providing these features, other key elements that of the design include support for assigning and tracking of individual message IDs, macro wrappers for log messages to allow groups of messages to be compiled in or out depending on the requirements of the build, and design of guaranteed and best-effort logging destinations. The next section covers the details of these features, how they will be implemented, and the performance issues. However, simply providing the APIs is not enough to ensure the maximum benefit is achieved from their use. There needs to be consistent use of the library throughout the entire code base. A later section will describe some guidelines.

# 3. Logging Architecture

This section describes the architecture of PVLogger along with some details on the behavior. The goal is to describe the conceptual framework and defined behaviors. Later sections will deal with APIs, implementation specifics, and performance issues. PVLogger consists of the following major components:

- Logging control nodes (a.k.a. Loggers)
- Logging destinations (a.k.a. Appenders)
- Message Filters
- Message formatters

The logging control nodes, or loggers, are the control points for determining, at runtime, which messages are logged via the log level. A hierarchical string, referred to as its tag, identifies each logger. For the tag, each level in the hierarchy is separated by a '.' (i.e., a period character). Therefore any text character, with the exception of a period character, can be used as a tag substring. The ABNF of the tag string is:

**Hierarchical Tag ABNF**

```
tag  = level-tag *(additional-levels)
additional-levels = "." level-tag
level-tag = 1*<tag-chars>
tag-chars = <US-ASCII 32-126 except '.' (octet 46)>
```

The only exception to the tag format described by the previous ABNF is the empty string, which is the universal root tag (i.e., the empty string is defined to be the prefix of every valid tag). ***Therefore the logger identified by the empty string is an ancestor of all loggers.*** These hierarchical tags impose a tree structure on the logger nodes. An example is shown in Figure 1.

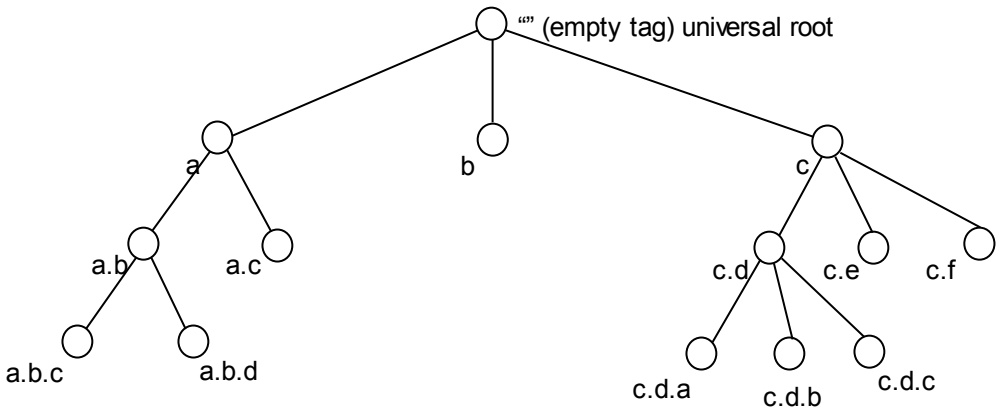


**Figure 1: Example Logger Tree Structure**

The structure in Figure 1 demonstrates how the logger tags introduce a natural descendant/ancestor structure, which will be used for determining the active log level for each node in the tree as well as the logging destinations (described later). The descendant/ancestor relationship between any two logger nodes is determined by the following rule:

**Descendant/Ancestor rule:**

***A logger node, L, is a descendant of another node, A, if and only if the tag for A is a prefix of the tag for L. Equivalently, A is said to be an ancestor of L.***

Messages are logged by passing the information to instances of the logger nodes along with a message level. The logger node determines if the message is active by comparing the message level with its active log level. A message is active if its message level is **less than or equal** to the active log level of the logger. To better understand the relationship, think of the log level as representing the volume of information so a higher log level means more messages being output in general. A message with a small message level value will tend to be output even when the log level and thus the volume of information is lower. Each logger can have an independent log level or inherit the log level of an ancestor logger. Once a log level is set in a node, it becomes the final inherited value for the subtree of its descendant nodes.

**Rule for determining the active log level of a logger:**

***The log level of a logger instance, L, is equal to the level specifically set for that control node or the inherited log level from the closest ancestor with a log level set.***

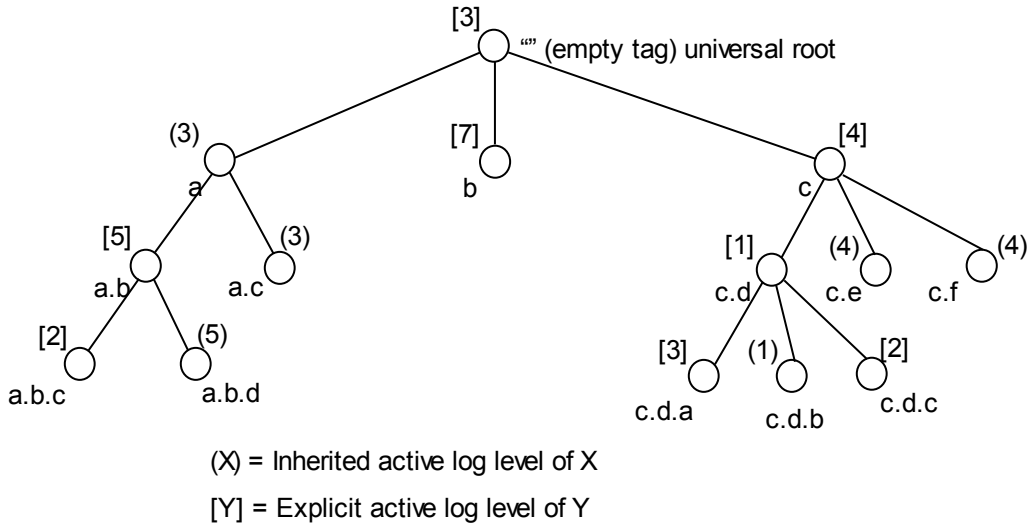


**Figure 2: Example of logger tree showing log level inheritance.**

Figure 2 shows an example logger tree with the associated active log level for each node. Some nodes have explicit log levels set while others inherit the value of an ancestor. For example, the node a.c inherits the log level of 3 from the root node while a.b has an explicit value of 5 set.

## 3.1. Appenders

While the loggers determine if a particular message is active or not, it's the appenders that are responsible for actually writing the message to the corresponding destination. Therefore, appenders are associated with specific log destinations (e.g., log files, memory buffers, serial ports, remote log servers, etc). A single appender can be attached to multiple logger nodes in arbitrary locations in the logger tree. Also each logger node holds a list of attached appenders. Once a logger has determined a message is active, it will send the message to all its attached appenders as well as all appenders inherited from ancestor nodes. It is possible to prevent the inheritance of appenders in a logger node by turning off the appender inheritance flag for that logger.

**Appender inheritance rule:**

***A logger inherits all the appenders of its ancestors up to the first ancestor node with the appender inheritance flag disabled.***

If the same appender is attached to multiple logger nodes in the same subtree, then a single message may be logged multiple times by the appender. Therefore, appenders are generally attached to disjoint subtrees to avoid duplication of log messages. Figure 3 shows an example of the logger tree along with the attached and inherited appenders. The example shows cases where appenders are attached to disjoint sections of the tree such as the serial appender attached to nodes a.b, c.d.a, and c.d.c. Also notice that since node c has the appender inheritance disabled and no appenders attached, no messages from logger c would ever be logged.

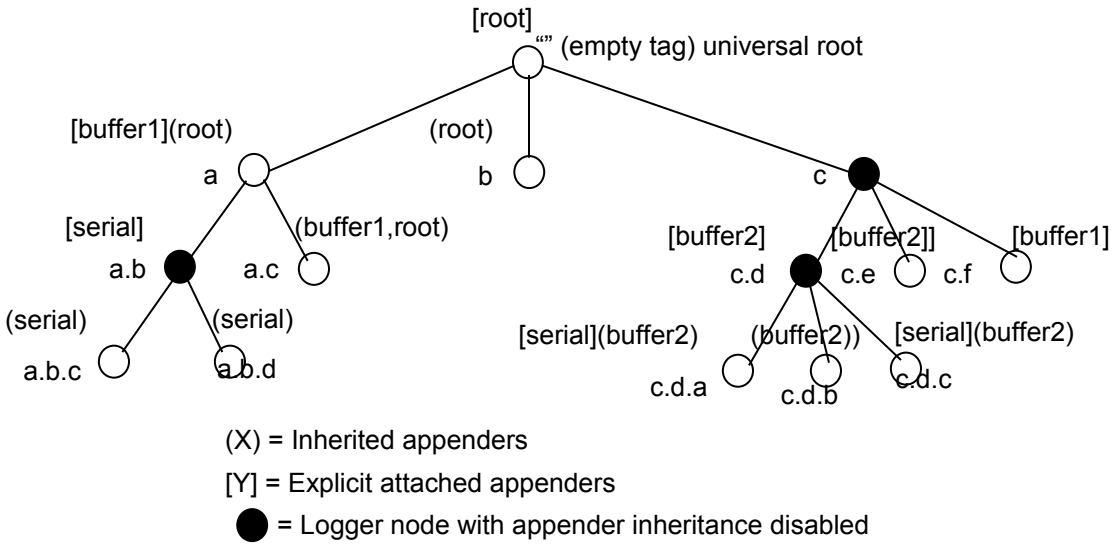


**Figure 3: Example logger tree showing appender inheritance.**

## 3.2. Message Filters

Message filters allow additional control over the messages logged by an appender. These filters can be chained together into a sequence and each appender can have an independent chain of message filters. The filter is given information on the logger tag, message ID, and message level to use in making a decision to accept or reject the message. The filter can also return a neutral decision in which case the next filter in the chain is run. The first filter to either accept or reject the message terminates the filter chain evaluation with the message handled according to that decision. If there are no filters attached to an appender or all filters return neutral, then the message is accepted by default.

## 3.3. Message Formatters

The appenders will rely on the attached message formatter to render the message information in its final format before it is written to the log destination. The formatter allows control over the layout of various parts of the message. For example: the timestamp format, order of fields (e.g., should message ID come before message level, etc), presence of certain fields, and format (e.g., packed binary, text, etc).

The combination of these four components (loggers, appenders, message filters, and message formatters) provides a very flexible system for categorizing and controlling messages. The hierarchical logger tags and the corresponding tree structure do not have to be completely determined at compile-time. Instead, the tags can be constructed at runtime if it's desirable to include some dynamic information in the logger context.

# 4. Logger APIs

A macro interface has been provided as the main method of writing log messages. The macros will have a look and feel similar to actual function calls, but they provide the following benefits:

- An easy way to compile-out log messages and completely remove any runtime performance and code size impacts.
- A way to avoid the overhead of argument evaluation when the log message is not active.

There are basically two APIs provided for logging individual messages: one focused on logging text messages and one focused on memory dumps (i.e., simply outputting the values in memory). The intended use of the text-based messages is convenient, human-readable formats without any need for post-processing. The binary messages are intended for minimizing performance impacts by allowing messages to be formatted and output as quickly as possible. Some post-processing will likely be required for binary messages. The form of the log message macros is shown in Figure 4 below.

The first three arguments to the macro are involved in determining whether the message is active or not. The instrumentation layer is used at compile-time to determine whether the message should be compiled-out or not. The use of the instrumentation layer value will be discussed in a later section. The next two arguments are used to determine if the message is active at run-time. The logger instance is used to access the set or inherited log level for the control node associated with this message. The message log level is the assigned level of this message that will be

compared with the active log level of the logger to determine if the message should be passed to the appenders. The remaining parameters are grouped together as a single macro argument, which will be written as the entire argument list of a function that will process the message if it's active. Both the text and binary macros take variable argument lists to include in the message. The variable argument lists are nice because they offer tremendous flexibility in creating messages with a very small number of APIs. The overhead to process the variable argument for the binary API should be quite reasonable, but if performance issues arise because of this overhead, it's possible to introduce another API with a fixed number of arguments. However the fixed argument API will only be considered as a final alternative if no other solutions exist to the performance issues.

**Text-based API:**

```
PVLOGGER_LOGMSG(I, C, L, (M, "format str", a, b, c, d, …));
```

where:

I = instrumentation layer

C = logging control point

L = message level

M = message ID

Format str = format string in the same format as taken by printf.

**Binary API:**

```
PVLOGGER_LOGBIN(I, C, L, (M, NumPairs, ptr1, len1, …));
```

The main difference compared with the text API is the NumPairs argument instead of a format string and the list of pointer and lengths values for memory blocks to be logged. The NumPairs argument is an integer that gives the number of pointer and length pairs that follow.

**Figure 4: Logger API macros.**

# 5. Performance Considerations

Performance considerations are an important part of every software design, but they are especially important for PVLogger because of its use in so many places. The design provides a number of ways to manage performance impacts. The first is the compile-time enabling and disabling of messages based on the instrumentation layer specified in the build. Messages can be organized so that groups can be compiled out as the build moves from a debug build to a release build. Each step of way, removes another instrumentation layer and reduces the performance impact. It's even possible to disable all messages to completely remove the logging impact.

At run-time, the individually controllable logger levels allow better control to only enable verbose logging for areas of interest. Similarly the flexible appender assignment and message filtering also help focus processing on only the messages of interest. Binary messaging can be used to minimize the message formatting cost.

Another performance consideration is the amount of effort needed to obtain a logger node based on its tag. This will need to be done in order to log any messages with that logger node. A map data structure will be used to hold the association between tags and logger nodes. Since this data structure is basically a balanced tree of nodes holding the tag value, the lookup complexity for an arbitrary node is $O(\log_2 N)$ where N is the number of elements in the map. The map will be built up at run-time as new nodes are requested. Additional complexity containment can be accomplished by limiting the number of logger nodes. To reduce the complexity logger nodes the number of lookup operations to obtain specific logger nodes should be minimized. The recommended approach is to obtain the relevant logger object during initialization and then use the same object for an extended period.

# 6. A Strategy for the Structured Use of Instrumentation and Logging

To get the maximum benefit from PVLogger some common strategies need to be followed throughout the code base. One of the most important is a consistent way of assigning instrumentation layers and logging levels to the messages. It's useful to think of logging beyond the scope of simply development debugging. There is a spectrum of uses ranging from low-level debug to medium and high-level debug to profiling and finally up to release / support diagnostic. The concept is illustrated in Figure 5.

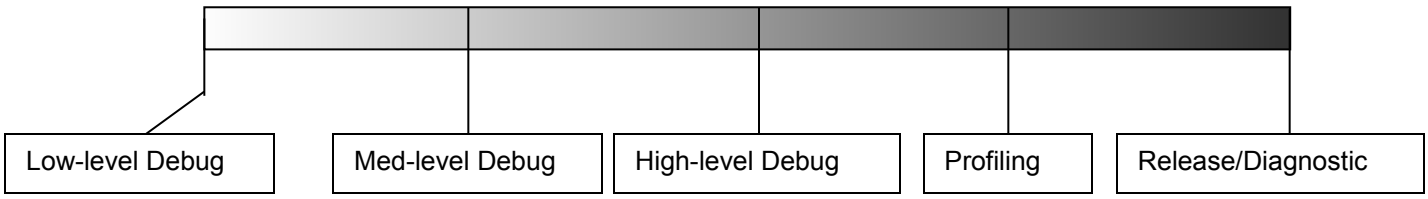


**Figure 5: Spectrum of uses for logging.**

There will tend to be a general correspondence between the position of the code in the software hierarchy and the type of messages, in terms of position on the logging/instrumentation spectrum, that appear there. The low-level libraries will tend to have a focus closer to the development debug end of the spectrum while the higher-level code will have a focus near the support and profiling end of the spectrum. The low-level debugging tends to be more concerned with timing independent logic while the profiling and release end of the spectrum is focused more on timing-related information and is much sensitive to the timing and performance impacts of logging. Each module or library should have a consistent and structured plan for placing messages at the appropriate instrumentation layer.

The unique message ID that is part of each logged message provides an opportunity for improved logging support. Today the code is not really taking advantage of this parameter and is simply setting that parameter to 0. The message ID of 0 is reserved by convention to mean “unset”. However if the message IDs are utilized, they do provide a way to quickly obtain a great deal of context information about the source of the message and are extremely useful for writing message filters. When using the message IDs, it is important that they are all unique and not reused for a different message (at least without some period where the message is deprecated). That is where a central repository of message IDs and related context information would be extremely useful. Each module or library could be assigned a range of message IDs to manage independently. This localizes the problem of assigning message IDs and makes it a bit easier. It also allows the module or library to be quickly identified from the ID based on the assigned ranges. Within each module, message IDs would be assigned whenever new messages are added or existing ones changed in any way. The correspondence between message ID and message content should be invariant once the message is introduced. Any new message IDs would be added to the central repository. The central repository of messages could then be used to answer a number of useful questions/queries:

- Show all messages deprecated in a specific release.
- Show all messages introduced in a specific release
- Find the first release that had a specific message ID
- Show messages in a specific module at a specific instrumentation layer.

# Appendix A: Example Code

The following code snippet shows the PVLogger-related API calls for creating an appender and enabling logging. It is not intended to be a complete listing of a module so steps such as the required initialization of OSCL are not shown here. The code example is from the sample player application, but similar examples can also be found in test code.

```
        ...
        // create an appender place in ref-counted container object
        appender = new StdErrAppender<TimeAndIdLayout, 1024>();
        OsclRefCounterSA<LogAppenderDestructDealloc<StdErrAppender<TimeAnd
IdLayout, 1024> > > *appenderRefCounter = new OsclRefCounterSA<
                    LogAppenderDestructDealloc<
                      StdErrAppender<
                            TimeAndIdLayout, 1024>>>(appender);
        refCounter = appenderRefCounter;
        OsclSharedPtr<PVLoggerAppender> appenderPtr(appender, refCounter);


        // set default log level to ERROR level
        // by setting the root level
        PVLogger *rootnode = PVLogger::GetLoggerObject("");
        rootnode->AddAppender(appenderPtr);
        rootnode->SetLogLevel(PVLOGMSG_ERROR);

        // log the PVPlayerEngine tag at DEBUG level
        PVLogger *node = PVLogger::GetLoggerObject(“PVPlayerEngine”);
        rootnode->AddAppender(appenderPtr);
        rootnode->SetLogLevel(PVLOGMSG_DEBUG);
```

# Appendix B: Partial List of Logger Tags

This section contains a partial list of logger tags currently present in the OpenCore code base. The list is may not be comprehensive but should provide a good starting point.

## Datapath Tag Hierarchy

The **datapath** tag hierarchy is related to information in the player that relates to the handling of media data as it travels through the flow graph. The descendant tags of the datapath allow focusing on specific nodes in the graph.

```
datapath
datapath.decnode
datapath.decnode.aacdecnode
datapath.decnode.m4vdecnode
datapath.sinknode
datapath.sinknode.in
datapath.sinknode.out
datapath.socketnode
datapath.sourcenode
datapath.sourcenode.aacparsernode
datapath.sourcenode.amrparsernode
datapath.sourcenode.jitterbuffer
datapath.sourcenode.jitterbuffer.flowctrl
datapath.sourcenode.jitterbuffer.fw
datapath.sourcenode.jitterbuffer.in
datapath.sourcenode.jitterbuffer.out
datapath.sourcenode.jitterbuffer.rtcp
datapath.sourcenode.medialayer
datapath.sourcenode.medialayer.in
datapath.sourcenode.medialayer.out
datapath.sourcenode.medialayer.portflowcontrol
datapath.sourcenode.mp4parsernode
datapath.sourcenode.protocolenginenode
```

## Pvplayerdiagnostics Tag Hierarchy

The pvplayerdiagnostics tag hierarchy is used to log statistics related to media processing from different components involved in playing media.

```
pvplayerdiagnostics
pvplayerdiagnostics.decnode.OMXAudioDecnode
pvplayerdiagnostics.decnode.OMXVideoDecnode
pvplayerdiagnostics.mp4ffparser
pvplayerdiagnostics.mp4parsernode
pvplayerdiagnostics.perf.engine
pvplayerdiagnostics.socketnode
pvplayerdiagnostics.streamingmanager
pvplayerdiagnostics.streamingmanager.medialayer
pvplayerdiagnostics.syncutil
```

## Pvauthordiagnostics Tag Hierarchy

The pvauthordiagnostics tag hierarchy is used to log statistics related to media processing from different components involved in authorng.

```
pvauthordiagnostics
pvauthordiagnostics.composer.mp4
pvauthordiagnostics.encnode.amrencnode
pvauthordiagnostics.encnode.h263encnode
pvauthordiagnostics.encoder.avc
pvauthordiagnostics.mio.aviwav
```

## Other Tags

The following is a sample of other tags found throughout the code. It is not meant to be comprehensive. Check the code of the components of interest for the precise set of tags.

```
Clock
clock.jitterbuffer
clock.jitterbuffernode
clock.jitterbuffernode.rebuffer
clock.streaming_manager.sessionduration
fmudiagnostics
HTTPRequest
JitterBuffer
JitterBufferNode
mp4ffparser
mp4ffparser_mediasamplestats
mp4ffparser_mediasamplestats_traf
mp4ffparser_parseddata
mp4ffparser_parseddata_traf
mp4ffparser_traf
OsclAsyncFile
oscldns
Oscl_File
OsclFileCache
OsclFileStats
osclib
OsclNativeFile
OsclSchedulerPerfStats
osclsocket
osclsocketserv
osclsocket_serv
OsclSocketStats
protocolenginenode.protocolengine
pvaacparser
```