Your Friend the System Log

This thread has been locked by a moderator.

1.7k

The unified system log on Apple platforms gets a lot of stick for being 'too verbose'. I understand that perspective: If you're used to a traditional Unix-y system log, you might expect to learn something about an issue by manually looking through the log, and the unified system log is *way* too chatty for that. However, that's a small price to pay for all its other benefits.

This post is my attempt to explain those benefits, broken up into a series of short bullets. Hopefully, by the end, you'll understand why I'm best friends with the system log, and why you should be too!

If you have questions or comments about this, start a new thread and tag it with *OSLog* so that I see it.

Share and Enjoy
—
Quinn "The Eskimo!" @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

Your Friend the System Log

Apple's unified system log is very powerful. If you're writing code for any Apple platform, and especially if you're working on low-level code, it pays to become friends with the system log!

The Benefits of Having a Good Friend

The public API for logging is fast and full-featured.

And it's particularly nice in Swift.

Logging is fast enough to leave log points [1] enabled in your release build, which makes it easier to debug issues that only show up in the field.

The system log is used extensively by the OS itself, allowing you to correlate your log entries with the internal state of the system.

Log entries persist for a long time, allowing you to investigate an issue that originated well before you noticed it.

Log entries are classified by subsystem, category, and type. Each type has a default disposition, which determines whether that log entry is enable and, if it is, whether it persists in the log store. You can customise this, based on the subsystem, category, and type, in four different ways:

- Install a configuration profile created by Apple (all platforms).
- Add an `OSLogPreferences` property to your app's `Info.plist` (all platforms).
- Run the `log` tool with the `config` command (macOS only)
- Create and install a custom configuration profile with the `com.apple.system.logging` payload (macOS only).

When you log a value, you may tag it as private. These values are omitted from the log by default but you can configure the system to include them. For information on how to do that, see Recording Private Data in the System Log.

The Console app displays the system log. On the left, select either your local Mac or an attached iOS device. Console can open and work with log snapshots (`.logarchive`). It also supports surprisingly sophisticated searching. For instructions on how to set up your search, choose Help > Console Help.

Console's search field supports copy and paste. For example, to set up a search for the subsystem `com.foo.bar`, paste `subsystem:com.foo.bar` into the field.

Console supports saved searches. Again, Console Help has the details.

Console supports viewing log entries in a specific timeframe. By default it shows the last 5 minutes. To change this, select an item in the Showing popup menu in the pane divider. If you have a specific time range of interest, select Custom, enter that range, and click Apply.

The `log` command-line tool lets you do all of this and more from Terminal.

There's a public API to read back existing log entries, albeit one with significant limitations on iOS (more on that below).

Every sysdiagnose log includes a snapshot of the system log, which is ideal for debugging hard-to-reproduce problems. For more information about sysdiagnose logs, see the info on Bug Reporting > Profiles and Logs.

But you don't have to use sysdiagnose logs. To create a quick snapshot of the system log, run the `log` tool with the `collect` subcommand. If you're investigating recent events, use the `--last` argument to limit its scope. For example, the following creates a snapshot of log entries from the last 5 minutes:

```
% sudo log collect --last 5m
```

For more information, see:

- os > Logging
- OSLog
- `log` man page
- `os_log` man page (in section 3)
- `os_log` man page (in section 5)
- WWDC 2016 Session 721 Unified Logging and Activity Tracing

[1] Well, *most* log points. If you're logging thousands of entries per second, the *very* small overhead for these disabled log points add up.

Foster Your Friendship

Good friendships take some work on your part, and your friendship with the system log is no exception. Follow these suggestions for getting the most out of the system log.

The system log has many friends, and it tries to love them the all equally. Don't abuse that by logging too much. One key benefit of the system log is that log entries persist for a long time, allowing you to debug issues with their roots in the distant past. But there's a trade off here: The more you log, the shorter the log window, and the harder it is to debug such problems.

Put some thought into your subsystem and category choices. One trick here is to use the same category across multiple subsystems, allowing you to track issues as they cross between subsystems in your product. Or use one subsystem with multiple categories, so you can search on the subsystem to see all your logging and then focus on specific categories when you need to.

Don't use too many unique subsystem and context pairs. As a rough guide: One is fine, ten is OK, 100 is too much.

Choose your log types wisely. The documentation for each `OSLogType` value describes the default behaviour of that value; use that information to guide your choices.

Remember that disabled log points have a very low cost. It's fine to leave chatty logging in your product if it's disabled by default.

No Friend Is Perfect

The system log API is hard to wrap. The system log is so efficient because it's deeply integrated with the compiler. If you wrap the system log API, you undermine that efficiency. For example, a wrapper like this is very inefficient:

```
// -*-*-*-*-*- DO NOT DO THIS -*-*-*-*-*-

void myLog(const char * format, ...) {
    va_list ap;
    va_start(ap, format);
    char * str = NULL;
    vasprintf(&amp;str, format, ap);
    os_log_debug(sLog, "%s", str);
    free(str);
    va_end(ap);
}

// -*-*-*-*-*- DO NOT DO THIS -*-*-*-*-*-
```

This is mostly an issue with the C API, because the modern Swift API is nice enough that you rarely need to wrap it.

If you do wrap the C API, use a macro and have that pass the arguments through to the underlying `os_log_xyz` macro.

iOS has very limited facilities for reading the system log. Currently, an iOS app can only read entries created by that specific process, using `.currentProcessIdentifier` scope. This is annoying if, say, the app crashed and you want to know what it was doing before the crash. What you need is a way to get all log entries written by your app (r. 57880434).

Xcode 15 beta has a shiny new console interface. For the details, watch WWDC 2023 Session 10226 Debug with structured logging.

In older versions of Xcode the console pane was not a system log client (r. 32863680). Rather, it just collected and displayed `stdout` and `stderr` from your process. This approach had a number of consequences:

- The system log does not, by default, log to `stderr`. Xcode enabled this by setting an environment variable, `OS_ACTIVITY_DT_MODE`. The existence and behaviour of this environment variable is an implementation detail and not something that you should rely on.
- Xcode sets this environment variable when you run your program from Xcode (Product > Run). It can't set it when you attach to a running process (Debug > Attach to Process).
- Xcode's Console pane does not support the sophisticated filtering you'd expect in a system log client.

When I can't use Xcode 15, I work around the last two by ignoring the console pane and instead running Console and viewing my log entries there.

If you don't see the expected log entries in Console, make sure that you have Action > Include Info Messages and Action > Include Debug Messages enabled.

The system log interface *is* available within the kernel but it has some serious limitations. Here's the ones that I'm aware of:

- This is no subsystem or category support)-:
- There is no support for annotations like `{public}` and `{private}`.
- Adding such annotations causes the log entry to be dropped (r. 40636781).

Revision History

- **2023-06-06** Updated to reference WWDC 2023 Session 10226. Added some notes about the kernel's system log support.
- **2023-03-22** Made some minor editorial changes.
- **2023-03-13** Reworked the Xcode discussion to mention `OS_ACTIVITY_DT_MODE`.
- **2022-10-26** Called out the Showing popup in Console and the `--last` argument to `log collect`.
- **2022-10-06** Added a link WWDC 2016 Session 721 Unified Logging and Activity Tracing.
- **2022-08-19** Add a link to Recording Private Data in the System Log.
- **2022-08-11** Added a bunch of hints and tips.
- **2022-06-23** Added the *Foster Your Friendship* section. Made other editorial changes.
- **2022-05-12** First posted.

OSLog

Reply

Posted 1 year ago by eskimo

Add a Comment

This site contains user submitted content, comments and opinions and is for informational purposes only. Apple disclaims any and all liability for the acts, omissions and conduct of any third parties in connection with or related to your use of the site. All postings and use of the content on this site are subject to the Apple Developer Forums Participation Agreement.

Copyright © 2023 Apple Inc. All rights reserved.