Logging System

The Engine includes a flexible, lightweight logging system designed for high-performance applications. It provides multiple levels of verbosity, supports compile-time removal of log calls to reduce binary size, and is built to be easily extended with features like file I/O and multi-threading.

The core design funnels all log messages through a single output function, which is then wrapped by a series of convenient preprocessor macros for ease of use. This approach offers granular control over what information is logged in different build configurations (e.g., Debug vs. Release).

Core Concepts: Log Levels

The logging system defines six distinct levels to categorize messages based on their severity and purpose. This allows developers to filter output and focus on the information that is most relevant to a given task.

LOG_LEVEL_FATAL

For critical errors that prevent the application from continuing. A fatal log indicates that the application must terminate.

LOG_LEVEL_ERROR

For serious errors that may not crash the application but leave it in a degraded or incorrect state. The application might recover, but its behavior will be unreliable.

LOG_LEVEL_WARN

For non-critical issues or sub-optimal conditions that do not prevent the application from running correctly but should be brought to the developer's attention.

LOG_LEVEL_INFO

For general, informational messages that are useful for tracking the application's lifecycle and state.

LOG_LEVEL_DEBUG

For detailed information intended for debugging purposes. These messages are only compiled in debug builds.

LOG_LEVEL_TRACE

The most verbose level, providing highly detailed, low-level information for intensive debugging. Like DEBUG, these messages are only compiled in debug builds.

C

```c
typedef enum log_level {
    LOG_LEVEL_FATAL = 0,
    LOG_LEVEL_ERROR = 1,
    LOG_LEVEL_WARN  = 2,
    LOG_LEVEL_INFO  = 3,
    LOG_LEVEL_DEBUG = 4,
    LOG_LEVEL_TRACE = 5,
} log_level;
```

Features and Compile-Time Configuration

A key feature of the logging system is the ability to enable or disable log levels at compile time. This ensures that unnecessary logging calls are completely removed from the final executable in release builds, eliminating any performance overhead.

Fatal and Error logs are always enabled. Other levels can be toggled using the following macros:

C

```
#define LOG_WARN_ENABLED 1
#define LOG_INFO_ENABLED 1
#define LOG_DEBUG_ENABLED 1
#define LOG_TRACE_ENABLED 1
```

For release builds, DEBUG and TRACE logs are automatically disabled by checking for the KRELEASE macro, ensuring that verbose debugging information never ships with the final product.

C

```
#if KRELEASE == 1
#define LOG_DEBUG_ENABLED 0
#define LOG_TRACE_ENABLED 0
#endif
```

API and Usage

The logging system is primarily accessed through a set of intuitive macros. These macros wrap the core log_output function, automatically providing the correct log level.

Core Functions

- b8 initialize_logging();
  - Sets up the logging system. In the future, this will handle tasks like opening a log file.
- void shutdown_logging();
  - Performs cleanup and ensures any queued log entries are written to disk.
- void log_output(log_level level, const char* message, ...);
  - The internal function that processes all log messages. Not intended to be called directly.

Logging Macros

These macros accept a printf-style format string and a variable number of arguments.

→ KFATAL(message, ...):
Logs a fatal message.

→ KERROR(message, ...):
Logs an error message.

→ KWARN(message, ...):
Logs a warning message (if enabled).

→ KINFO(message, ...):
Logs an informational message (if enabled).

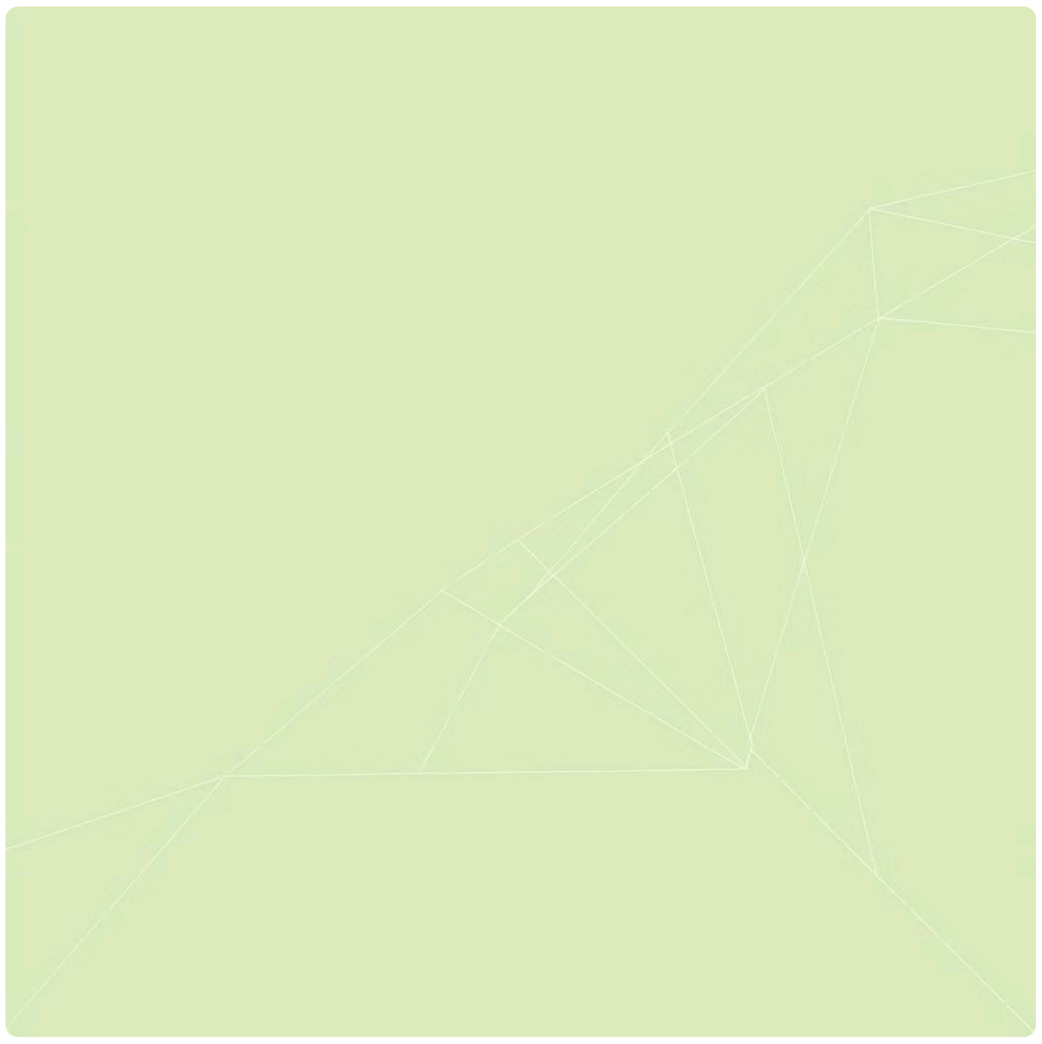
→ KDEBUG(message, ...):
Logs a debug message (if enabled and in a debug build).

→ KTRACE(message, ...):
Logs a trace message (if enabled and in a debug build).

Implementation Details

The implementation of log_output in logger.c is designed for efficiency and portability.

- **Static Buffer**: It uses a large, stack-allocated character array (32000 characters) to format log messages. This avoids the performance cost of dynamic memory allocation (malloc) for every log call.
- **Variadic Arguments**: It uses __builtin_va_list and the <stdarg.h> library (va_start, vsnprintf, va_end) to handle printf-style formatting safely. This is done to maintain compatibility with the Clang compiler, which may have issues with MSVC's va_list typedef.
- **Message Formatting**: The final output message is constructed in two steps. First, a prefix string corresponding to the log level (e.g., [FATAL]:) is prepended to the message. Then, the user-provided message and arguments are formatted before it is printed to the console.



C

```c
// Example snippet from log_output
void log_output(log_level level, const char* message, ...) {
    const char* level_strings[6] = {"[FATAL]: ", "[ERROR]: ", ...};

    char out[32000];
    memset(out, 0, sizeof(out));

    size_t prefix_len = snprintf(out, sizeof(out), "%s", level_strings[level]);
    prefix_len = (prefix_len >= sizeof(out)) ? sizeof(out) - 1 : prefix_len;


    __builtin_va_list arg_ptr;
    va_start(arg_ptr, message);
    size_t msg_len = (size_t)vsnprintf(out+ prefix_len, sizeof(out) - prefix_len, message, arg_ptr);
    msg_len = (msg_len >= sizeof(out) - prefix_len) ? (sizeof(out) - prefix_len) - 1 : msg_len;
    va_end(arg_ptr);

    // ... (ensure proper null-termination, append newline, and handle platform-specific output)
}
```