Log Facility for C

2016/03/12	0.1	Initial Version	Muhammad Z
2016/03/18	0.2	High Level Macros removed	Robert P
2016/03/25	0.9	Implementation details	Muhammad Z
2016/03/29	0.99	Final Draft	Muhammad Z
2016/04/14	1.0	Version 1	Muhammad Z

Summary

The purpose of this project is to create a log facility for use from C programs.
You will gain experience in:

1. Module and API Design
2. Advanced C programming
3. Putting Data Structures to actual usage.
4. Macros

What is a log?

A log can be a file or a stream where we can store log entries. Log entry contains information that depicts an error message or a trace message. Naturally, a program composed of different modules will have different requirements for logging. While some modules will be writing only critical errors other modules might be configured to output very detailed trace messages to help understand a situation.

Basically any module can be specifically configured to write to a specific log file only entries of certain level or higher.

What is a log entry?

A log entry is basically a record (terminated with a new line) containing a detailed information written into the log file. The record contains information provided by the developer and information gathered automatically. Each entry has a given severity. If the log entry severity is lower than the current log level it will be ignored otherwise it will be written to the log file.

Log Entry Detailed Requirements

A log entry written to the log file will look differently depending on the build type:

Debug builds:
If NDEBUG is not defined when building the code using the log facility, then an entry will look like:

```
 +- year
 |   +- month              +- process id
 |   |  +- day             |      +- thread id
 |   |  |                  |      |
2016-4-29  22:43:20.244  40059  1299 I query dequeue@reader.c:9 Queued tasks: 4
            |                        |  |     |      |        | +-- message w/ params
            +- time                  |  |     |      |        +--- line number
                                     |  |     |      +- source file name
                                     |  |     +- function name
                                     |  +- tag - Module name
                                     +- log level (first letter)
```

Where the message parts in blue are provided by the programmer and the rest generated automatically.

Release Builds
If NDEBUG is defined then the log will not have: file name, function name or line number.

Requirements

R1. Efficiency

The Log will be efficient for use from single threaded applications. Nevertheless, there will be minimal performance overhead penalty for calling log functions that will not actually write a log entry due to current configuration.

R2. Log Levels

The Log will support various levels of logging. It will support these levels with an option to extend to more fine grained levels:

LOG_**T**RACE	Trace message usually very detailed
LOG_**D**EBUG	A message useful for debugging
LOG_**I**NFO	Informative message
LOG_**W**ARNING	Warning
LOG_**E**RROR	An error occurred
LOG_**C**RITICAL	Critical error
LOG_**S**EVERE	Server error requiring immediate intervention
LOG_**F**ATAL	Fatal error signaling an emergency
LOG_NONE	Used only in configuration file

If a the current log level configured for a module is higher than the log level associated with a log entry, then that log entry will not be written to the log file. The bold letter (blue) is what appears in the log entry in the log file.

Each log level has an associated priority. LOG_NONE is highest level and will be used only in configuration file to silence log entries from a given module.
LOG_TRACE is lowest priority and effectively will allow all log entries to written to log file.

R3. Log Configuration File

The Log facility will be configured by a simple configuration file. The configuration file will have exactly one mandatory section of general settings and multiple per module sections with specific setting for each module not using the general settings.

A. General section
   A special named section - starting with [#] - appearing at the beginning of the configuration file specifying general log setting.
   1. Default log level, only log messages with equal or higher level will actually be saved to the log file.
   2. Directory path specifying where all log files will be placed.
   3. File name to be used for logging entries.

```
[#]
Level = default_log_level
Path = log_base_directory
File = default_log_file_name
```

B. Module section, specifying log settings for each individual configured module. There can be multiple module sections in the file following the general section:
   Each section will specifying :
   1. Module tag name - a short module name or tag
   2. Current log level for this module, only log messages with equal or higher level will actually saved to the log file.
   3. File name to be used for logging entries tagged with this module tag, the file will be created under the base directory specified in the general section.

```
[Module_Tag_Name]
Level = log_level
File = log_file_name
```

Usage Example

A program is composed of the following modules:

1. **net** : network manager
2. **store** : file storage manager
3. **hash** : Hash-table based in memory database
4. **query** : query processor

Log Configuration File

A sample log configuration file:

```
[#]
Level = LOG_ERROR
Path = /var/log/dbServer
File = db.log

[net]
Level = LOG_WARNING
File = network.log

[query]
Level = LOG_DEBUG
File = querry_processor.log
```

Using these setting:

1. All log files will be place under /var/log/dbServer directory
2. Logging from **net** module will be:
    a. Written out if they have level of `LOG_WARNING` or above
    b. Saved to *`/var/log/dbServer/network.log`*
3. Logging from **query** module will be:
    a. Written out if they have level of `LOG_DEBUG` or above
    b. Saved to *`/var/log/dbServer/querry_processor.log`*
4. Logging from **store, hash** and any other module will use the general setting, hence they will be:
    a. Written out if they have level of `LOG_ERROR` or above
    b. Saved to *`/var/log/dbServer/db.log`*

Initial setup of log facility

The log facility should be initialized during the application startup. Usually in main() :

```
#include "zlog4c.h"
int main(int argc, char* argv[]){
     zlog_init("app.log.config");
     ...
}
```

Usage from a module

The use of the log facility from a module, first you need to get a log object for the said module. Then you use ZLOG macro to send messages to the module specific log file. If the log was not configured for this module, the general settings will be used. Otherwise module specific configurations holds.

For instance:

```
     #include "zlog4c.h"

     #define MODULE "net"     /* network module */
     int net_start_listening(int port){
(1)       ZLog log = zlog_get(MODULE);
(2)       ZLOG(log, LOG_TRACE, "will start listening on port %d", port);
          ...
          if( r != 0 ){
(3)            ZLOG(log, LOG_ERROR, "listen on port:%d failed: %d", port, r);
               return r;
          }
          ...
```

(1) - Initialize or get the logger for module "net"
(2) - This log entry will **not** be recorded since it's level is lower than the configured level for this module (LOG_TRACE < LOG_ERROR). See sample configuration file above.
(3) - This log entry will be recorded according to the above configuration file.

Deliverables

1. Static library: libzlog4c.a
2. Header file: zlog4c.h
   This header file is intended for users of the library. It defines all required macros, functions and types exposed to the consumer of the library.

Documentation

The zlog4c.h header file should be fully documented using doxygen style comments. Cover the macros, public functions and the log levels.

Constraints

- The log library will handle error in the logging process gracefully. It will report internal errors (of it's own to standard error stream or to the general log file if available)
- If no log was initialized then the default behavior will be to log LOG_ERROR and above to standard error.
- The log library is not required to be multithreaded safe in this version.
- Log entries will be appended to the log file if it exists, it's not supposed to truncate existing files.
- Make sure the shutdown process is graceful.

Implementation Hints

- zlog_init function will never crash the process.
- zlog_get function will never fail, if requested log name is not defined, it will return a universal placeholder that will send log entries marked as error or above to standard error.
- ZLOG is a variadic macro defined as
  #define ZLOG(LOG, LEVEL, fmt, …) *IMPLEMENTATION DETAILS*
  Where:

  LOG is the log obtained via zlog_get

  LEVEL is one of LOG_TRACE … LOG_FATAL levels

  fmt is a format string as used by printf

  … is for accepting variable number of arguments matching the fmt string
- ZLOG macro will not execute any computation if current log level for module does not allow for the entry to be written. i.e:

  ZLOG(moduleLog, LOG_INFO, “Signal Power %lf “, getSpectrumPower());

  If current log level is above LOG_TRCAE, no log entry will be added and getSpectrumPower Will not be called.
- Use *snprintf* function for formatting, use a buffer of 1024 bytes.
- Since variadic macros are not covered by ANSI C standard, we need to silence compiler error message by using a pragma directive:

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wvariadic-macros"

#define ZLOG(......) .....details.....

#pragma GCC diagnostic pop
```