Logger API Usage

Logger is a light, simple, multiple-level, configure-able utility to record log to standard output, file and remote machine designed by Ryan Powell (ryan@gorillaapps.co.uk). First, it should be initialized with customized configuration files and default log level. Second, users can uses kinds of API to record log in different level, last the DE-construction should be called.

Currently, this utility has been expanded from the original code base from git lab , the enhancement includes:

- Add TRACE level;
- Add C function and demo to replace default printf/fprintf
- Add two micro to initialize Logger fast to simplify its usage
- Fix the header file name error in logger_pluginStdout.c
- Add SConstruct/SConscritp support to auto compile source code and test/example demos
- Add support to compile with DEBUG mode and remove some warnings
- Replace "LOGGER" with "LOG" to reduce typing
- Add one script to support log level change during running and add TODO to record TODO list later

To help user understand the API usage quickly, following instructions are highly recommend to be known first.

1. Initialization

1.1 prepare the configuration file

First, create a new configuration file for your specific case. You can refer demos below for quick start.

Demo 1: logger configuration file to let it print to standard output

```
[root@localhost test]# pwd
/root/20170317-0900/lib/Logger/test
[root@localhost test]# ls
test_ini.ini  test_logger_output.c  test_logger_output.h  test_logger_output.o  test_main  test_main.c  test_main.o  test_run.sh
[root@localhost test]# cat test_ini.ini
[output=stdout]
[root@localhost test]#
```

Demo 2: logger configuration file to let it print to specified file

```
[root@localhost libtest]# pwd
/root/20170317-0900/lib/Logger/libtest
[root@localhost libtest]# ls *.ini
libtest.ini
[root@localhost libtest]# cat libtest.ini
#[output=my_logger.txt]
#[output=file] output=/home/qxia/20161224-2150/lib/Logger/example/my_logger.txt
[output=file] output=./print_logger.txt
```

Second, use should call loggerLoadIniFile() to load the configuration file in the code, demo code is as below:

```
int main(int argc, const char * argv[])
{
    if ( argc != 2 )
    {
        fprintf(stderr, "Must be called with ini file to load (%u)\n",argc);
        return 1;
    } else {
        char *inifile = (char*)argv[1];

        fprintf(stdout, "Loading ini file: %s\n",inifile);

        /* load the configuration file which details which files have what logging permissions */
        loggerLoadIniFile(inifile, (uint32_t)strlen(inifile));
```

Demo 3:

Or you can follow sample code below to init LoggerLIB:

*static char * inifile = "/etc/libcommand_logger.ini";*
static LOG_OUTPUT_HANDLE _loggerHandle = NULL;

*static void logger_init(char * confile)*
{

assert(NULL != confile);

INIT_LOGGER_LIB(_loggerHandle, confile);

LOG_ENTRY;
}

1.2 set log level
currently 10 different kinds of log level are supported, they are list as below:

```
typedef enum _LOG_LEVEL
{
    LOG_LEVEL_NONE    = 0U,
    LOG_LEVEL_ENTRY   = 1U,
    LOG_LEVEL_EXIT    = 2U,
    LOG_LEVEL_TRACE   = 4U,
    LOG_LEVEL_INFO    = 8U,
    LOG_LEVEL_WARN    = 16U,
    LOG_LEVEL_ERROR   = 32U,
    LOG_LEVEL_FATAL   = 64U,
    LOG_LEVEL_ASSERT  = 128U,
    LOG_LEVEL_EVENT   = 256U,
} LOG_LEVEL;
```

priority of log increases as its default enum value increases. The higher priority a log owns, the stronger requirement it has to post the log messages. Fox example, user can use LOGLIB_ENABLE_TYPE(LOG_LEVEL_TRACE) to allow all output from

LOG_TRACE()/LOG_INFO()/LOG_WARN()/LOG_ERROR()/LOG_FATAL()/LOG_ASSERT()/LOG_EVENT() to be printed to standard output or specified file according to configuration file. Vice versa, if user find too many output from LOG_TRACE() and want to disable log in TRACE level,
LOGLIB_DISABLE_TYPE(LOG_LEVEL_TRACE) can be applied, then all output from LOGLIB_ENTRY()/LOGLIB_EXIT()/LOGLIB_TRACE() won't be disabled.

Currently, default log level of this library are set as below:
LOG_LEVEL_INFO | LOG_LEVEL_WARN | LOG_LEVEL_ERROR | LOG_LEVEL_FATAL | LOG_LEVEL_EVENT

```
static LOG_LEVEL f_defaultLevel = LOG_LEVEL_INFO | LOG_LEVEL_WARN | LOG_LEVEL_ERROR | LOG_LEVEL_FATAL | LOG_LEVEL_EVENT;
```

That means if this step is skipped, output of LOG_TRACE() will be disabled.

1.3 call log output API

Demo code is as below:

LOGLIB_ERROR("%s", "Prepare initGroups failure!\n");
…….
LOGLB_ERROR("Cmd doesn't match usage rule!\n");

Will output is as bellowing:

17:09:55 14/04/17|initiator_test.c|715|init_initGrp|ERROR|Prepare initGroups failure!

17:09:55 14/04/17|initiator_test.c|1123|common_op_wrappers|ERROR|Cmd doesn't match usage rule!

2. frequently used API of various log level

[root@localhost inc]# pwd
/root/20170411-1656/lib/Logger/inc

#define LOGLIB_INIT   loggerInitFromFileName(logger_get_handle_addr(), __FILE__, strlen(__FILE__))
#define LOGLIB_TERM   loggerTerm(logger_get_handle())
#define LOGLIB_ENABLE_TYPE(a) loggerAppendDebugLevel(logger_get_handle(), (a))
#define LOGLIB_DISABLE_TYPE(a) loggerRemoveDebugLevel(logger_get_handle(), (a))

#define LOGLIB_ENTRY LOG_PRINT_ENTRY(logger_get_handle(), NULL)
#define LOGLIB_EXIT LOG_PRINT_EXIT(logger_get_handle(), NULL)

#define LOGLIB_TRACE(format, ...) LOG_PRINT_TRACE(logger_get_handle(), format, ##__VA_ARGS__)
#define LOGLIB_INFO(format, ...) LOG_PRINT_INFO(logger_get_handle(), format, ##__VA_ARGS__)
#define LOGLIB_WARN(format, ...) LOG_PRINT_WARN(logger_get_handle(), format, ##__VA_ARGS__)
#define LOGLIB_ERROR(format, ...) LOG_PRINT_ERROR(logger_get_handle(), format, ##__VA_ARGS__)
#define LOGLIB_FATAL(format, ...) LOG_PRINT_FATAL(logger_get_handle(), format, ##__VA_ARGS__)
#define LOGLIB_ASSERT(format, ...) LOG_PRINT_ASSERT(logger_get_handle(), format, ##__VA_ARGS__)
#define LOGLIB_EVENT(format, ...) LOG_PRINT_EVENT(logger_get_handle(), format, ##__VA_ARGS__)

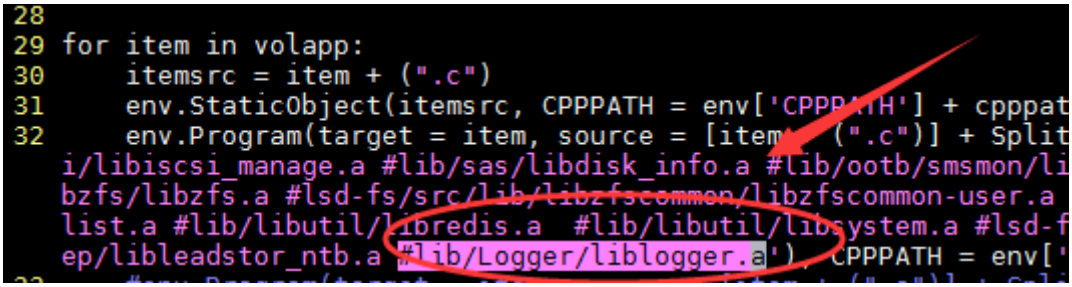
[root@localhost inc]# ls *.h
loggerFacade.h logger.h

4. Compiling

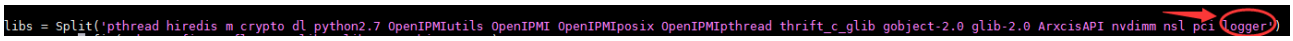
step 1: include the header file
add : #include <loggerFacade.h>
step 2: Update CPPFLAGS
add “'#lib/Logger/inc” to cpppath
step 3: link static library or dynamic library

To use static library:

add “#lib/Logger/liblogger.a” to linker:



To use dynamic library: add “logger” to libs



5. WARNING

- Ensure the initialization of logger library should be executed only once in your

executable binary! It is recommend to do the initialization in main() of every progress.

- Don't initialize lib Logger in you code which aims to generate static or dynamic library