
RT-THREAD ULOG Log Component

Application Notes - Basics

RT-THREAD Document Center

Copyright ©2019 Shanghai Ruisaide Electronic Technology Co., Ltd.



WWW.RT-THREAD.ORG

Tuesday 9th October, 2018

Table of contents

Table of contents	i
1 Purpose and structure of this paper.	1
1.1 Purpose and Background of this Paper	1
1.2 Structure of this paper.	1
2 Problem description.	1
3. Problem solving.	2
3.1 Introduction to ulog.	2
3.2 Overview of the ulog framework.	2
3.3 Configuration instructions.	3
3.4 Log tags.	4
3.4.1. Use tags to ensure log modularity.	4
3.4.2. How to define a tag	4
3.5 Log levels.	5
3.5.1. Setting Level Classification	5
3.5.2. How to use the log output API.	6
3.5.3. Output raw logs.	7
3.5.4. Use in interrupt ISR.	7
3.6 Assertions.	8
3.7 Set the log format.	8
3.8 hexdump	10
4 Frequently Asked Questions.	11
5 References.	11
5.1 All APIs related to this article.	11

- 5.1.1. API List 11
- 5.1.2. Detailed explanation of the core API. 11
- 5.1.3. ulog initialization. 11
- 5.1.4. ulog deinitialization. 12
- 5.1.5. LOG_X Log Output API 12
- 5.1.6. ulog_x log output API. 12
- 5.1.7. Output hex format log. 13

This application note introduces the basics of the RT-Thread ulog component and its basic usage, helping developers get started with it. For more advanced usage of the ulog component, see the RT-Thread ulog Logging Component Application Note - Advanced.

1 Purpose and structure of this paper

1.1 Purpose and Background of this Paper

Definition of a log: Logs record information about software running status and progress, output to various media (e.g., files, consoles, and displays), and store it for display and storage. They provide a reference for software debugging and maintenance, problem tracing, performance analysis, system monitoring, and fault warnings. Logs typically occupy at least 80% of the software lifecycle.

The importance of logging: Due to the immense complexity of operating systems, single-step debugging is not suitable in some scenarios. Therefore, logging components are almost always standard in operating systems. A comprehensive logging system can also make operating system debugging more efficient.

The origins of **ulog** : RT-Thread has long lacked a compact, practical logging component, and ulog was created to fill this gap. It will be open-sourced as a foundational component of RT-Thread, allowing developers to utilize a simple, easy-to-use logging system and improve development efficiency.

1.2 Structure of this paper

This application note will introduce the RT-Thread ulog component from the following aspects:

- Introduction to the ulog component and framework overview
- Configuration of the ulog component
- Use of the basic functions of the ulog component

2 Problem Description

This application note will introduce the RT-Thread ulog component around the following issues.

- What are the main functions of the ulog component?
- What are the commonly used logging interfaces?
- How to use ulog?

To solve these problems, you first need to understand the basic functions of the RT-Thread ulog component, then become familiar with the commonly used log API, and finally demonstrate how to use ulog on the qemu platform.

3. Problem Solving

3.1 Introduction to ulog

ulog is a very concise and easy-to-use C/C++ logging component. The first letter "u" stands for "micro." It can achieve a minimum resource usage of **<1KB of ROM and <0.2KB of RAM**. ulog is not only compact but also very comprehensive. Its design concept is based on another open source C/C++ logging library: EasyLogger (elog for short), and it has made many improvements in terms of functionality and performance. The main features are as follows:

- The backend for log output is diversified and supports backend forms such as serial port, network, file, flash memory, etc.;
- Log output is designed to be thread-safe and supports asynchronous output mode;
- The log system is highly reliable and can still be used in complex environments such as interrupt ISR and Hardfault;
- Supports dynamic/static switches to control the global log output level;
- The logs of each module support dynamic/static setting of output levels;
- Log content supports global filtering by keywords and tags;
- The API and log format are compatible with **Linux syslog**;
- Supports dumping debug data into the log in hex format;
- Compatible with rtdbg (RTT's early log header file) and EasyLogger's log output API.

3.2 Overview of the ulog framework

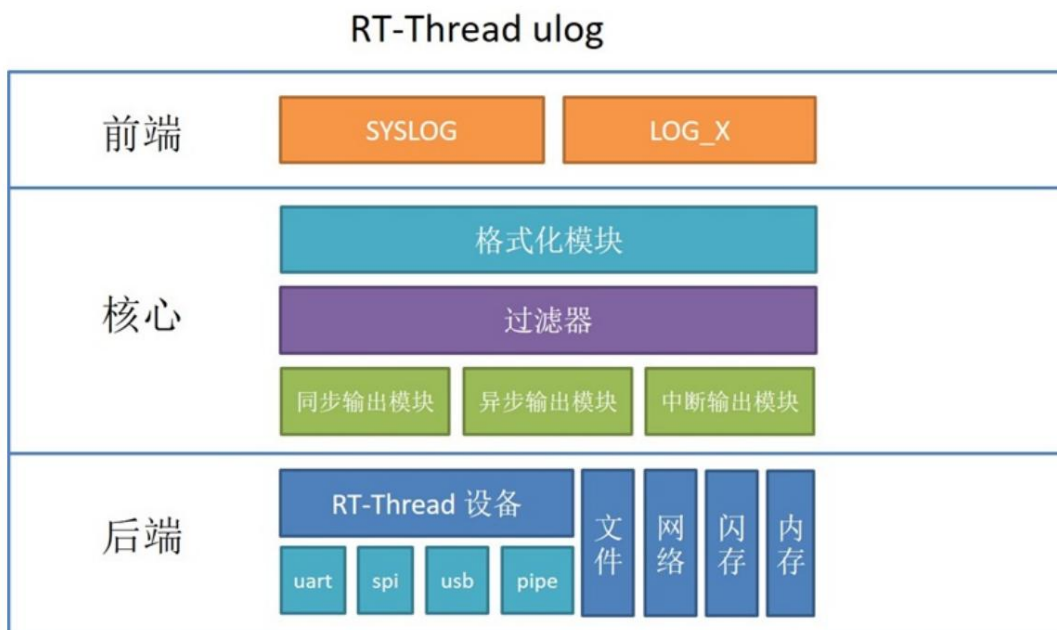


Figure 1: ulog frame

The above picture is the internal framework diagram of ulog, from which we can see:

- **Front-end:** This layer is the closest to the application and provides users with two types of API interfaces: syslog and LOG_X. Convenient for users to use in different scenarios;
- **Core:** The main task of the middle core layer is to format and filter the logs passed from the upper layer according to different configuration requirements and then generate log frames. Finally, through different output modules, they are output to the bottom-level backend devices;
- **Backend:** After receiving the log frame from the core layer, it outputs the log to the registered log backend device.

3.3 Configuration Instructions

Download the RT-Thread source code, use the env tool to enter the `rt-thread\bsp\qemu-vexpress-a9` folder, enter `menuconfig` to open the configuration menu, and you can see the ulog configuration item under **RT-Thread Components** → **Utilities**. Enable it and you will see the following configuration interface:

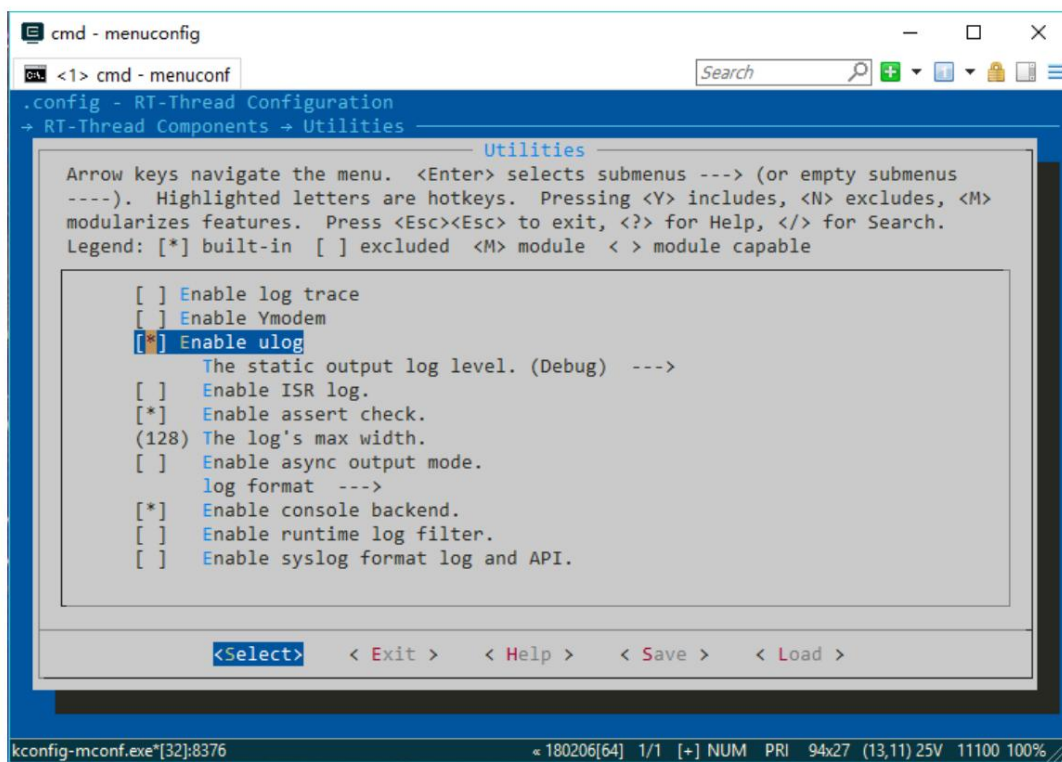


Figure 2: ulog Configuration

The configuration description of each option is as follows:

- **The static output log level. (Debug)** : Select the static log output level. After the selection is completed, logs with a lower level than the set level (here specifically logs using LOG_X API) will not be compiled into ROM.
- **Enable ISR log** : Enable interrupt ISR log, that is, the log output API can also be used in ISR.
- **Enable assert check** : Enable assertion check. When disabled, the assertion log will not be compiled into ROM.
- **The log's max width** : The maximum length of the log. Since the ulog log API is in lines, this length also represents the maximum length of a line of log.
- **Enable async output mode** : Enable asynchronous log output mode. After turning on this mode, the log will not be output to the backend immediately, but will be cached first and then handed over to the log output thread (for example, the idle thread) for output.

This mode has many benefits, which will be introduced in detail in the "RT-Thread ulog Log Component Application Notes - Advanced Edition".

- **log format** : configure the log format, such as time information, color information, thread information, whether to support floating

Points and so on.

- **Enable console backend** : Enable the console as the backend. After enabling, logs can be output to the console serial port. It is recommended to keep it enabled. • [Enable](#)

runtime log filter: Enable the runtime log filter, that is, dynamic filtering. After enabling, the log

It will support dynamic filtering by tags, keywords, etc. during system operation. • [Enable syslog](#)

format log and API : Enable the Linux syslog API and the corresponding log format.

Use the default configuration, save and exit menuconfig.

3.4 Log Tags

Tags are a common way of classification. Ulog also gives each log a tag attribute to facilitate classification management.

3.4.1. Using tags to ensure log modularity

As log output continues to increase, to avoid disorganized log output, each log needs to be categorized using tags in a modular manner. For example, logs from the Wi-Fi driver module use the tag [wifi.driver](#) , and logs from the Wi-Fi device management module use the tag [wifi.mgmt](#) .

The tag attributes of each log can also be output and displayed. At the same time, ulog can also set the output level of the log corresponding to each tag (module). The logs of currently unimportant modules can be selectively closed, which not only reduces ROM resources, but also helps developers filter irrelevant logs.

The output level of the log corresponding to each tag (module) also supports dynamic adjustment at runtime. For details, see "RT-Thread ulog Log Component Application Notes - Advanced Edition".

3.4.2. Label definition method

See the [rt-thread\examples\ulog_example.c](#) ulog example file, where **LOG_TAG** is defined at the top of the file.

Macro:

```
#define LOG_TAG          "example"          //The label corresponding to this module. If not defined,
    default: NO_TAG
#define LOG_LVL when     LOG_LVL_DBG //The log output level corresponding to this module.
    defined, default: debug level
#include <ulog.h>          //Must be under LOG_TAG and LOG_LVL
```

It should be noted that when defining the log tag, it must be above **##include <ulog.h>** , otherwise the default NO_TAG will be used (it is not recommended to define these macros in the header file).

Log tags are scoped to the current source code file. Project source code is typically categorized by module. Therefore, when defining tags, you can specify the module or submodule name as the tag name. This not only makes log output clear and intuitive, but also facilitates dynamic level adjustment or filtering based on tags.

3.5 Log Levels

The log level represents the importance of the log. In ulog, there are several log levels from high to low.

level	Name Description
LOG_LVL_ASSERT	asserts that an unhandled, fatal error has occurred, so that the system cannot continue to run. Assertion Log
LOG_LVL_ERROR	error: The log output when a serious, unrecoverable error occurs is an error-level log.
LOG_LVL_WARNING	Warnings are output when minor, repairable errors occur. Report Log
LOG_LVL_INFO	information is an important reminder information log for upper-level users of this module, such as: initial The level of log is usually retained during mass production.
LOG_LVL_DBG	debugs the debug log for the module developers to view. This level of log is usually used in mass production. closure

3.5.1. Classification of setting levels

In ulog, log levels can be divided into the following categories:

- Static and dynamic levels: Logs are categorized based on whether they can be modified at runtime. Logs with a lower level than the static level (specifically, logs using the LOG_X API) will not be compiled into ROM and ultimately not output or displayed. The dynamic level controls logs with a level higher than or equal to the static level. When ulog is running, logs with a lower level than the dynamic level will be filtered out.
- Global level and module level: These are categorized by scope. In ulog, each file (module) can also set an independent log level. The global level has a larger scope than the module level, meaning that the module level can only control module logs that are higher than or equal to the global level.

From the above classification, we can see that in ulog, the output level of the log can be set through the following 4 aspects:

- Global static log level: configured in menuconfig, corresponding to the `ULOG_OUTPUT_LVL` macro
- Global dynamic log level: use the `void ulog_global_filter_lvl_set(rt_uint32_t level)` function
Number to set
- Module static log level: define `LOG_LVL` macro in the module (file) and define the log tag macro `LOG_TAG`
Similar method
- Module dynamic log level: Use `int ulog_tag_lvl_filter_set(const char *tag, rt_uint32_t level)` function to set

Their scope of action is as follows:

Global static > Global dynamic > Module static > Module dynamic

Using the Log Output API

ulog has two main log output APIs

- `LOG_X("msg")` macro API: `X` corresponds to the first letter of the different levels, with the capital letters. The APIs are `LOG_D`, `LOG_E`, etc. This method is preferred because it has a simple API format and only takes one parameter, the log message. It also supports filtering by module static log level.
- `ulog_x("tag", "msg")` macro API: `x` corresponds to the abbreviation of different levels. This API is suitable for using different tags to output logs in a file.

3.5.2. How to use the log output API

The following will introduce the ulog routine. Open `rt-thread\examples\ulog_example.c` and you can see the label and static priority of the file defined at the top.

```
#define LOG_TAG          "example"
#define LOG_LVL          LOG_LVL_DBG
#include <ulog.h>
```

The `LOG_X` API is used in the `void ulog_example(void)` function, as follows:

```
/* output different level log by LOG_X API */
LOG_D("LOG_D(%d): RT-Thread is an open source IoT operating system from
      China.", count);
LOG_I("LOG_I(%d): RT-Thread is an open source IoT operating system from
      China.", count);
LOG_W("LOG_W(%d): RT-Thread is an open source IoT operating system from
      China.", count);
LOG_E("LOG_E(%d): RT-Thread is an open source IoT operating system from
      China.", count);
```

These log output APIs all support the printf format and will automatically wrap the log at the end.

The following will show the running effect of the ulog routine on qemu:

- Copy `rt-thread\examples\ulog_example.c` to `rt-thread\bsp\qemu-vexpress-a9\`

In the `applications` folder, go to

the `rt-thread\bsp\qemu-vexpress-a9` directory in the env directory. After confirming

that ulog configuration has been executed before, execute the `scons` command and wait for the compilation to complete. Run `qemu.bat` to open the qemu simulator of RT-

Thread. Enter the `ulog_example` command to see the results of the ulog routine. The general effect is as shown below

```

cmd - qemu.bat
Administrator@ARMINK E:\program\RTT_GCC\rt-thread\bsp\qemu-vexpress-a9
> qemu.bat
WARNING: Image format was not specified for 'sd.bin' and probing guessed raw.
        Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
        Specify the 'raw' format explicitly to remove the restrictions.
dsound: Could not initialize DirectSoundCapture
dsound: Reason: No sound driver is available for use, or the given GUID is not a valid DirectSound device ID

\ | /
- RT -   Thread Operating System
/ | \   3.1.1 build Sep 25 2018
2006 - 2018 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[I/SAL_SOC] Socket Abstraction Layer initialize success.
[I/[SDIO]] SD card capacity 65536 KB.
[I/[SDIO]] probe mmcblk0 block device!
hello rt-thread
msh />ulog_example
[6389] D/example: LOG_D(1): RT-Thread is an open source IoT operating system from China.
[6392] I/example: LOG_I(1): RT-Thread is an open source IoT operating system from China.
[6395] W/example: LOG_W(1): RT-Thread is an open source IoT operating system from China.
[6400] E/example: LOG_E(1): RT-Thread is an open source IoT operating system from China.
[6406] D/test: ulog_d(1): RT-Thread is an open source IoT operating system from China.
[6410] I/test: ulog_i(1): RT-Thread is an open source IoT operating system from China.
[6415] W/test: ulog_w(1): RT-Thread is an open source IoT operating system from China.
[6421] E/test: ulog_e(1): RT-Thread is an open source IoT operating system from China.
qemu-system-arm.exe[32]:16824
180206[64] 1/1 [++] NUM PRI: 116x27 (7,32766) 25V 14944 100%

```

Figure 3: *ulog* Routines

You can see that each log entry is displayed line by line, with different levels of logs displayed in different colors. At the beginning of the log is a tick for the current system, followed by the log level and tags, and at the end is the specific log content. These log formats and configuration instructions will be discussed later in this article.

3.5.3. Output raw logs

The output of APIs such as `LOG_X` and `ulog_x` are all formatted logs. Sometimes it is necessary to output logs without any formatting. When logging, you can use the `LOG_RAW` or `void ulog_raw(const char *format, ...)` function. For example:

```
LOG_RAW("\r");
ulog_raw("\033[2A");
```

3.5.4. Use in interrupt ISR

Many times, it is necessary to output logs in the interrupt ISR, but the interrupt ISR may interrupt the log output. To ensure that interrupt logs and thread logs do not interfere with each other, special processing must be performed for interrupt situations.

Ulog has integrated the interrupt log function, but it is not enabled by default. When using it, turn on the [Enable ISR log](#) option. The log API is the same as that used in the thread, for example:

```
#define LOG_TAG          "driver.timer"
#define LOG_LVL          LOG_LVL_DBG
#include <ulog.h>

void Timer2_Handler(void) {
```

```

/* enter interrupt */
rt_interrupt_enter();

LOG_D("I'm in timer2 ISR");

/* leave interrupt */
rt_interrupt_leave();
}

```

Here are the different strategies for interrupt logging when ulog is in synchronous mode and asynchronous mode:

- In synchronous mode: If a thread is interrupted while it is outputting logs, and there are logs to be output in the interrupt, Output will be directly output to the console and does not support output to other backends;
- In asynchronous mode: If the above situation occurs, the log in the interruption will be put into the buffer first, and finally added to the thread log. The logs are handed over to the log output thread for processing.

3.6 Assertions

ulog also provides an assertion API : `ASSERT(expression)` . When the assertion is triggered, the system will stop running and `ulog_flush()` will be executed internally, and all log backends will be flushed. If asynchronous mode is enabled, all logs in the buffer will also be flushed. The following is an example of assertion usage:

```

void show_string(const char *str) {

    ASSERT(str);
    ...
}

```

3.7 Setting the log format

The log formats supported by ulog can be configured in `menuconfig`, located in RT-Thread Components `Utilities` `ulog` `log` format. The specific configuration is as follows:

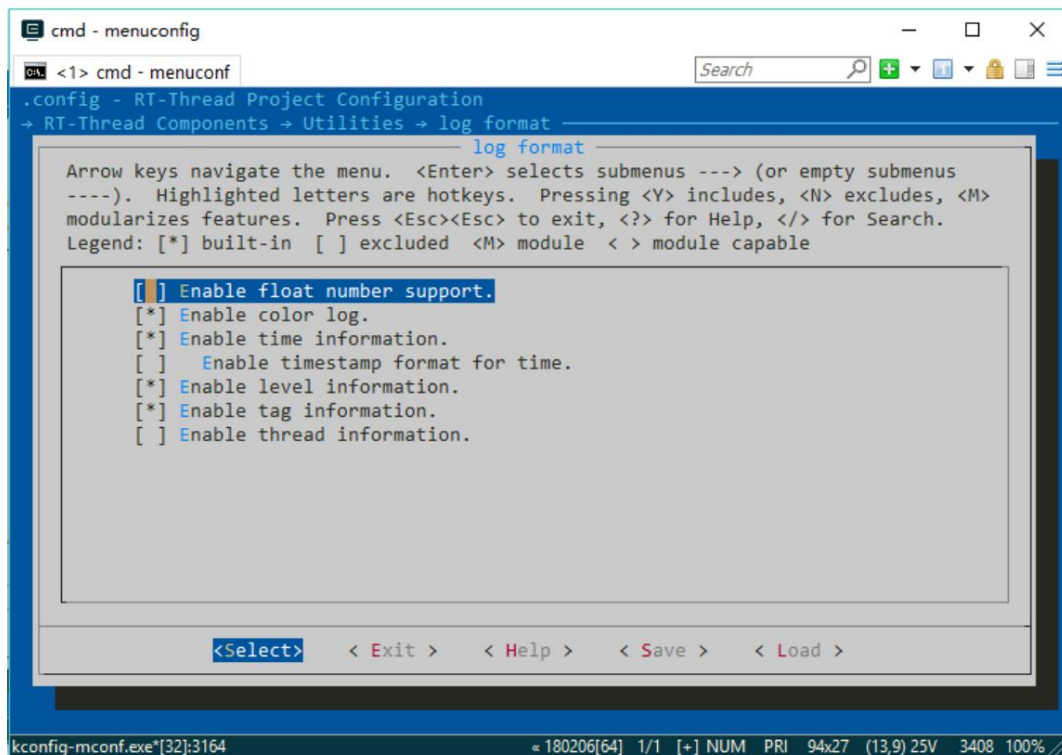


Figure 4: ulog Format Configuration

You can configure: floating point number support (traditional rtdebug/rt_kprintf do not support floating point logs), Color log, time information (including timestamp), level information, tag information, thread information. Select all the options, save, recompile and run the ulog routine again in qemu to see the actual effect:

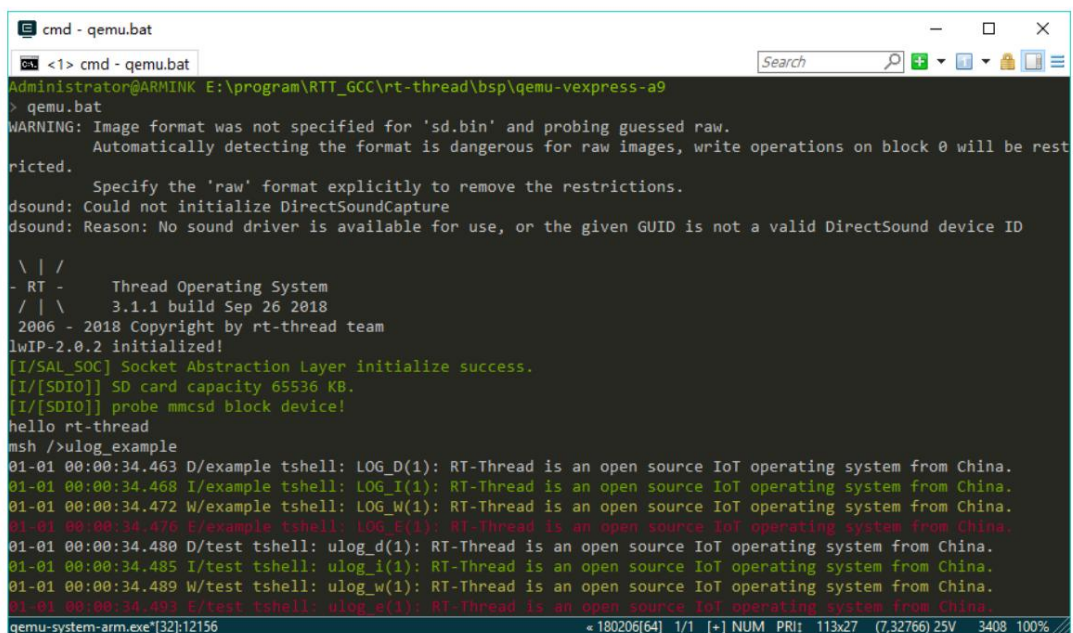


Figure 5: ulog Routines (All formats)

It can be seen that compared with the first time the routine was run, new timestamp information and thread information have been output.

3.8 hexdump

Hexdump is also a commonly used function for log output. It can be used to output a piece of data in hex format.

The corresponding API is: `void ulog_hexdump(const char *name, rt_size_t width, rt_uint8_t *buf, rt_size_t size)`. The following is a look at the specific usage and running effect:

```
/* Define an array of 128 bytes in length*/ uint8_t i,
buf[128]; /* Fill the array with
numbers*/

for (i = 0; i < sizeof(buf); i++) {

    buf[i] = i;

} /* Dump the data in the array in hex format, with a width of 16 */
ulog_hexdump("buf_dump_test", 16, buf, sizeof(buf));
```

You can copy the above code into the ulog routine and run it, and then see the actual running results:

```
cmd - qemu.bat
<1> cmd - qemu.bat
\ | /
- RT -      Thread Operating System
/ | \      3.1.1 build Sep 26 2018
2006 - 2018 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[I/SAL_SOC] Socket Abstraction Layer initialize success.
[I/[SDIO]] SD card capacity 65536 KB.
[I/[SDIO]] probe mmcblk0 block device!
hello rt-thread
msh />ulog example
D/HEX buf_dump_test: 0000-0010: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F .....
D/HEX buf_dump_test: 0010-0020: 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F .....
D/HEX buf_dump_test: 0020-0030: 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F !"#%&'()*+,-./
D/HEX buf_dump_test: 0030-0040: 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 0123456789:;<=>?
D/HEX buf_dump_test: 0040-0050: 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F @ABCDEFGHIJKLMNO
D/HEX buf_dump_test: 0050-0060: 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F PQRSTUVWXYZ[\]^_
D/HEX buf_dump_test: 0060-0070: 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F `abcdefghijklmnopqrstuvwxyz
D/HEX buf_dump_test: 0070-0080: 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F pqrstuvwxyz{|}~.
01-01 00:00:31.107 D/example tshell: LOG_D(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:31.112 I/example tshell: LOG_I(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:31.117 W/example tshell: LOG_W(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:31.122 E/example tshell: LOG_E(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:31.126 D/test tshell: ulog_d(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:31.131 I/test tshell: ulog_i(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:31.135 W/test tshell: ulog_w(1): RT-Thread is an open source IoT operating system from China.
01-01 00:00:31.140 E/test tshell: ulog_e(1): RT-Thread is an open source IoT operating system from China.
qemu-system-arm.exe*[32]:13824 180206[64] 1/1 [+] NUM PRI: 113x27 (7.32766) 25V 3408 100%
```

Figure 6: ulog Routines (hexdump)

It can be seen that the middle part is the hexadecimal information of the buf data, and the rightmost part is the character information corresponding to each data.

So far, the basics of ulog have been introduced. If you want to know more about the advanced use of ulog, you can

Continue to view "RT-Thread ulog log component application notes - advanced"

4 Frequently Asked Questions

- 1. The logging code has been executed, but there is no output

Refer to the Log Levels section to learn about log level classifications and check the log filter parameters. Another possibility is that you accidentally disabled the console backend. Re-enable [Enable console backend](#) to resolve the issue .

- 2. The end of the log content is missing

This is because the log content exceeds the maximum width of the log. Check the log's [max width](#) option and increase it to an appropriate size.

- 3. Why can't I see millisecond time after turning on timestamp?

This is because ulog currently only supports displaying millisecond timestamps when software emulation RTC is enabled. If you need to display it, just enable RT-Thread software emulation RTC function.

- 4. Every time before including the ulog header file, [LOG_TAG](#) and [LOG_LVL](#) must be defined . Can this be simplified?

If [LOG_TAG](#) is not defined, the [NO_TAG](#) tag will be used by default. This will easily cause misunderstandings in the output logs, so it is not recommended to omit the tag macro.

If [LOG_LVL](#) is not defined, the debug level will be used by default. If the module is in the development stage, this process can be omitted.

However, if the module code is already stable, it is recommended to define this macro and change the level to the information level.

5References

5.1 All related APIs in this article

API List

API	Location
int ulog_init(void)	ulog.c
void ulog_deinit(void)	ulog.c
LOG_E(...) / LOG_W(...) / LOG_I(...) / LOG_D(...) / LOG_RAW(...)	ulog.h
ulog_e(TAG, ...) / ulog_w(TAG, ...) / ulog_i(TAG, ...) / ulog_d(TAG, ...)	ulog_def.h
void ulog_hexdump(const char name, rt_size_t width, rt_uint8_t buf, rt_size_t size)	ulog.c

5.1.2. Detailed explanation of core API

5.1.3. ulog initialization

```
int ulog_init(void)
```

Before using ulog, you must call this function to complete ulog initialization. If the component automatic initialization is turned on, the API will also be called automatically.

return	describe
≥ 0	success
-5	Failed, out of memory

5.1.4. ulog deinitialization

```
void ulog_deinit(void)
```

When ulog is no longer used, you can execute deinit to release resources.

5.1.5. LOG_X Log Output API

```
LOG_X(...)
```

This API is a macro, and X corresponds to the capitalized first letter of different levels.

Note: Before using this API, you need to define `LOG_TAG` and `LOG_LVL` above the ulog.h header file.

Macro, see the section on using the log output API for details.

This API can be used to output logs of corresponding levels based on defined tags.

parameter	describe
...	Log content, the format is consistent with printf

5.1.6. ulog_x log output API

```
ulog_x(TAG, ...)
```

This API is a macro, and x corresponds to the lowercase first letter of different levels.

Note: Before using this API, you need to define the `LOG_LVL` macro above the ulog.h header file, see:

Chapter on using the log output API.

This API can specify tags when outputting logs of corresponding levels. It has one more input parameter than `LOG_X` and is not recommended.

parameter	describe
-----------	----------

TAG	Log Tags
-----	----------

...	Log content, the format is consistent with printf
-----	---

5.1.7. Output **hex** format log

```
void ulog_hexdump(const char *name, rt_size_t width, rt_uint8_t *buf,
                  rt_size_t size)
```

Dump data in hexadecimal format to the log. For details on usage and effects, see the hexdump section.

parameter	describe
-----------	----------

name	Log Tags
------	----------

width	The width of a line of hex content (number)
-------	---

buf	Data content to be output
-----	---------------------------

size	Data size
------	-----------
