RT-THREAD ULOG Log Component

Application Notes **-** Basics

RT-THREAD Documentation Center

WWW.RT-THREAD.ORG

Tuesday 9th October, 2018

Table of contents

This application note introduces the basic knowledge of RT-Thread ulog component and the basic usage of ulog, which helps developers to get started with RT-Thread ulog component. For more advanced usage of ulog component, please refer to "RT-Thread ulog log component application note - advanced version".

1 Purpose and structure of this paper

1.1 Purpose and Background of this Paper

Definition of log: Log is to output the software running status, process and other information to different media (such as files, consoles, display screens, etc.), and display and save them. It provides a reference for software debugging, problem tracing, performance analysis, system monitoring, fault warning and other functions during maintenance. It can be said that the use of logs occupies at least 80% of the software life cycle.

Importance of logs: For operating systems, due to the complexity of the software, single-step debugging is not suitable in some scenarios, so log components are almost standard on operating systems. A complete log system can also make operating system debugging more efficient.

The origin of **ulog :** RT-Thread has always lacked a small and practical log component, and the birth of ulog has made up for this shortcoming. It will be open sourced as a basic component of RT-Thread, so that our developers can also use a simple and easy-to-use log system to 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 Statement

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 log 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 APIs, 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++ log component. The first letter u stands for micro. It can achieve the minimum resource usage of **ROM<1K and RAM<0.2K .** ulog is not only small in size, but also has very comprehensive functions. Its design concept refers to another C/C++ open source log library: EasyLogger (elog for short), and has made many improvements in functions and performance. The main features are as follows:

- The backend for log output is diversified, supporting 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

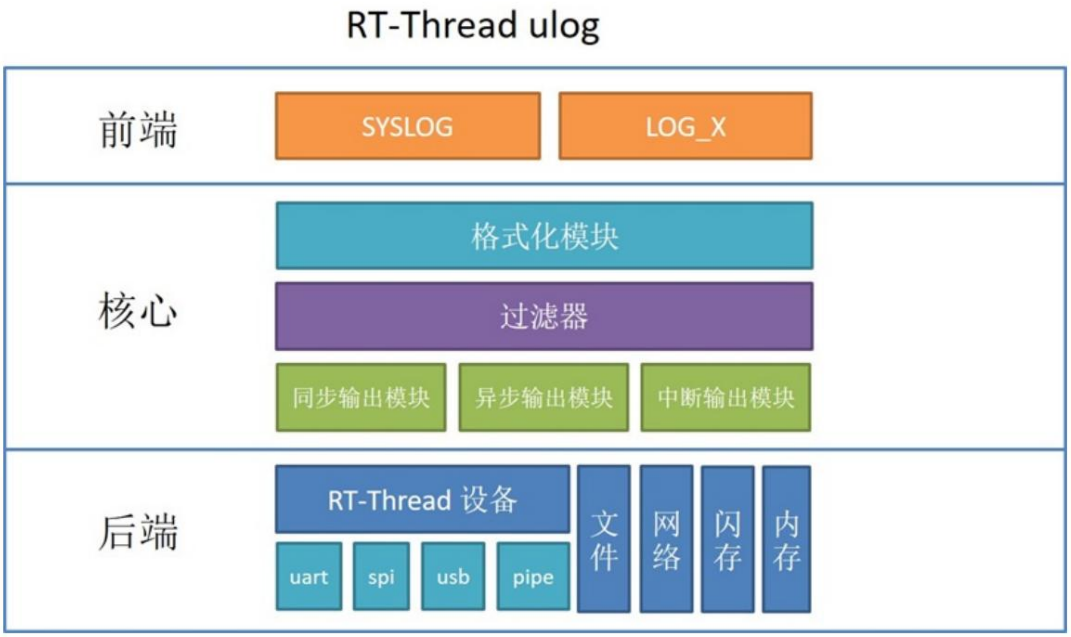

Figure **1:** *ulog* frame

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

- Front-end: As the layer closest to the application, this layer 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, which are finally output to the bottom-level backend device through different output modules;
- Backend: After receiving the log frame sent by 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 ÿ Utilities . Enable it and you can see the following configuration interface:

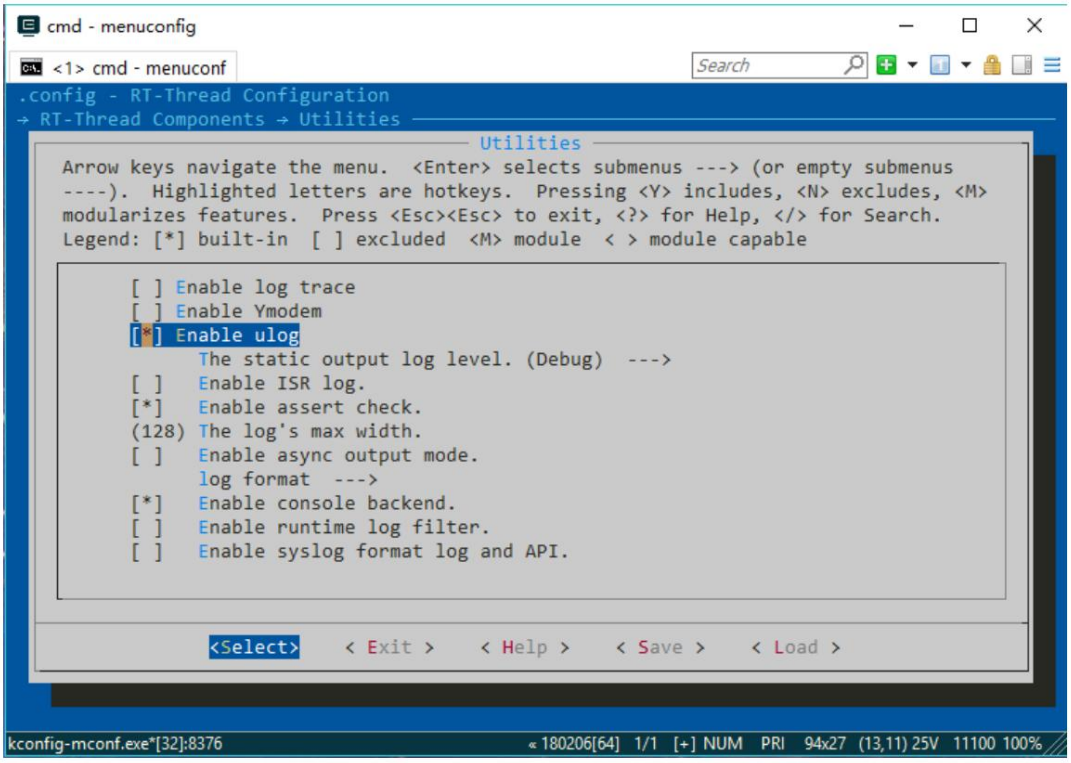

Figure **2:** *ulog* Configuration

The configuration description of each option is as follows:

- The **static** output log level. (Debug) ÿSelect 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 turned off, 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: idle thread) for output.

This mode has many benefits, which will be introduced in detail in "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, i.e. dynamic filtering. After enabling, logs It will support dynamic filtering by tags, keywords, etc. while the system is running. • 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. Use tags to ensure log modularity

As the amount of log output continues to increase, in order to avoid the logs being output in a disorderly manner, each log needs to be classified using tags in a modular way. For example, the log of the Wi-Fi driver module uses the wifi.driver tag, and the log of the Wi-Fi device management module uses wifi.mgnt as the tag.

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 rt-thread\examples\ulog_example.c ulog example file, at the top of the file there is a definition of LOG_TAG Macros:

```
#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).

The scope of the log tag is the current source code file, and the project source code is usually classified by module. Therefore, when defining a tag, you can specify the module name or submodule name as the tag name, which is not only clear and intuitive when displaying the log output, but also convenient for dynamic adjustment of levels or filtering by tags in the future.

3.5 Log Level

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, recoverable errors occur. Report Log
LOG_LVL_INFO	information is an important information log for upper-level users of this module to view, such as: initial The logs at this level are 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 level and dynamic level: Logs are classified according to 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 the ROM, and will not be output or displayed in the end. The dynamic level can control logs that are 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: are classified according to scope. In ulog, each file (module) can also set an independent log level. The global level scope is greater than the module level, that is, the module level can only control module logs that are higher than or equal to the global level.

Based on the above classification, we can see that in ulog, the output level of the log can be set in the following 4 aspects

- Global static log level: configured in menuconfig, corresponding to ULOG_OUTPUT_LVL macro •
  Global dynamic log level: use **void** ulog_global_filter_lvl_set(rt_uint32_t level) function
  Number to set
- Module static log level: define the 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 scopes are related as follows:

Global static > Global dynamic > Module static > Module dynamic

Use of 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 in capital letters, and the API is LOG_D , LOG_E , etc. This method is the first choice, on the one hand because its API format is simple, there is only one input parameter, namely the log information, and on the other hand, 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 that there are labels and static priorities defining the file 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 at the end of the log.

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\applications folder• Enter the rt-thread\bsp\qemu-vexpress-a9 directory in env• 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 emulator of RT-Thread• Enter the ulog_example command to see the results of the ulog routine running, the general effect is as shown below

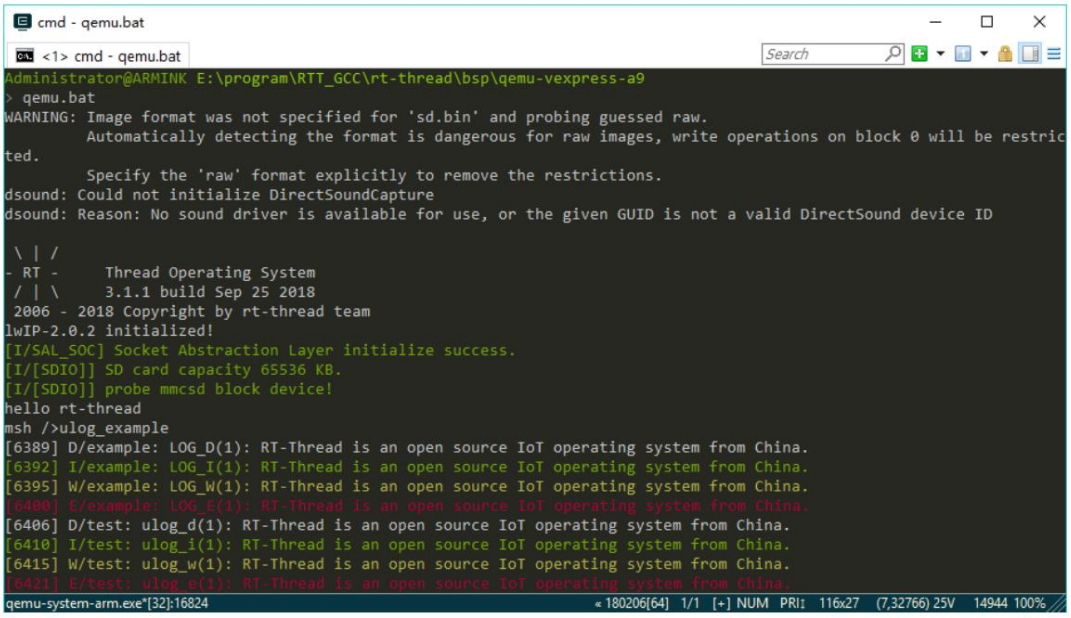

Figure **3:** *ulog* Routines

You can see that each log is displayed in lines, and different levels of logs have different colors. At the beginning of the log, there is a tick of the current system, in the middle there are log levels and tags, and at the end there are specific log contents. These log formats and configuration instructions will also be introduced in detail later in this article.

3.5.3. Output **raw** log

The output of APIs such as LOG_X and ulog_x are all formatted logs. Sometimes you need to output logs without any formatting. When logging, you can use 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, you need to output logs in the interrupt ISR, but the interrupt ISR may interrupt the process of outputting logs. 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, if there are logs to be output in the interrupt
  Output will be directly output to the console, and output to other backends is not supported;
- In asynchronous mode: If the above situation occurs, the log in the interruption will be put into the buffer first, and finally combined with the thread log.
  The log is 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, ulog_flush() will be executed internally , and all log backends will execute flush. If the asynchronous mode is turned on, all logs in the buffer will also be flushed. The following is an example of the use of assertions:

```
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 ÿ Utilities ÿ ulog ÿ log format. The specific configuration is as follows:

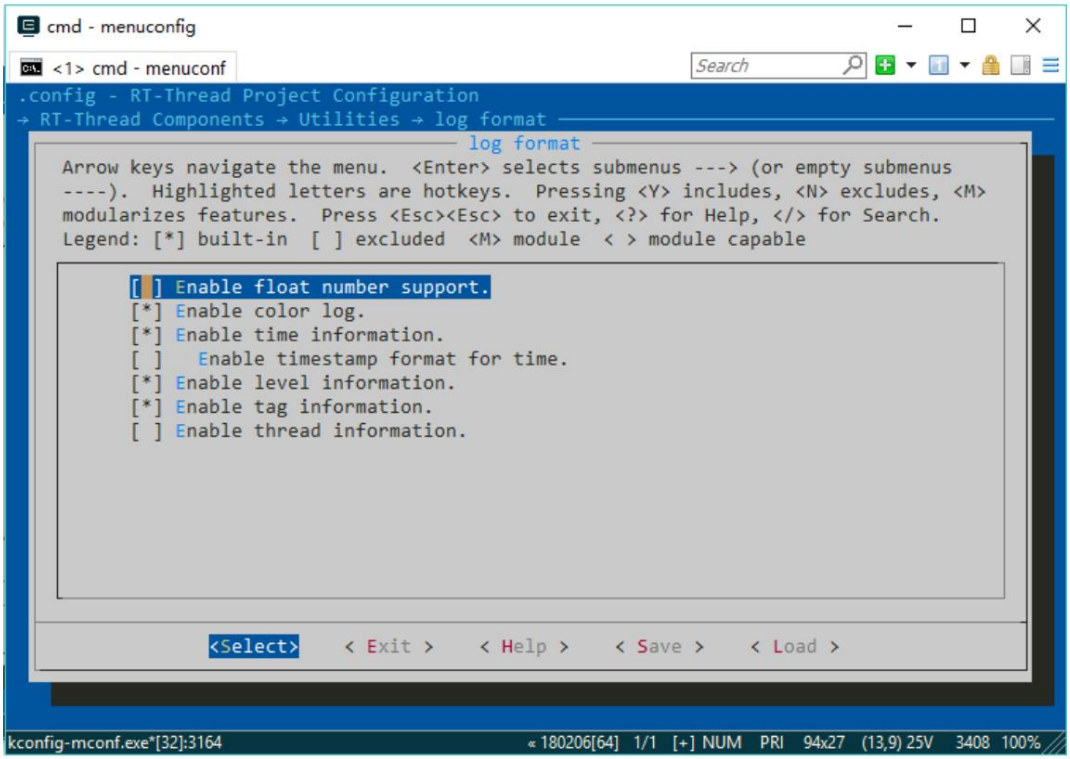

Figure **4:** *ulog* Format Configuration

You can configure: floating point number support (traditional rtdbg/rt_kprintf do not support floating point logs), Color logs, time information (including timestamps), level information, tag information, and thread information.

Select all the items, save, recompile and run the ulog routine again in qemu to see the actual effect:

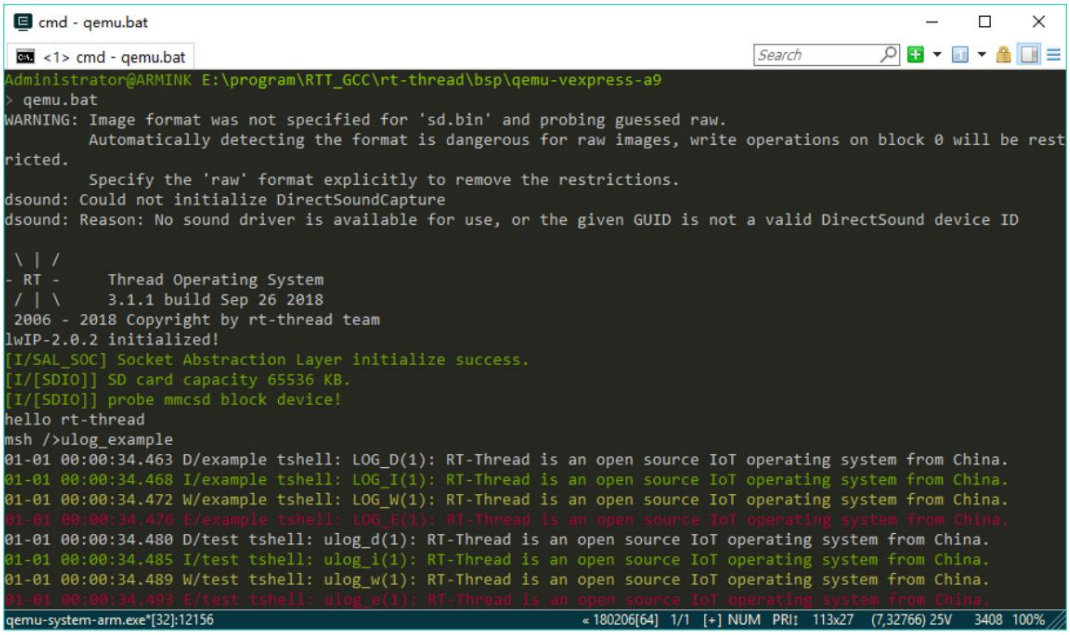

Figure **5:** *ulog* Routines *(*All formats *)*

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

3.8 hexdump

Hexdump is also a commonly used function for log output. It can 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) . Let's take 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, width is 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 look at the actual running results:

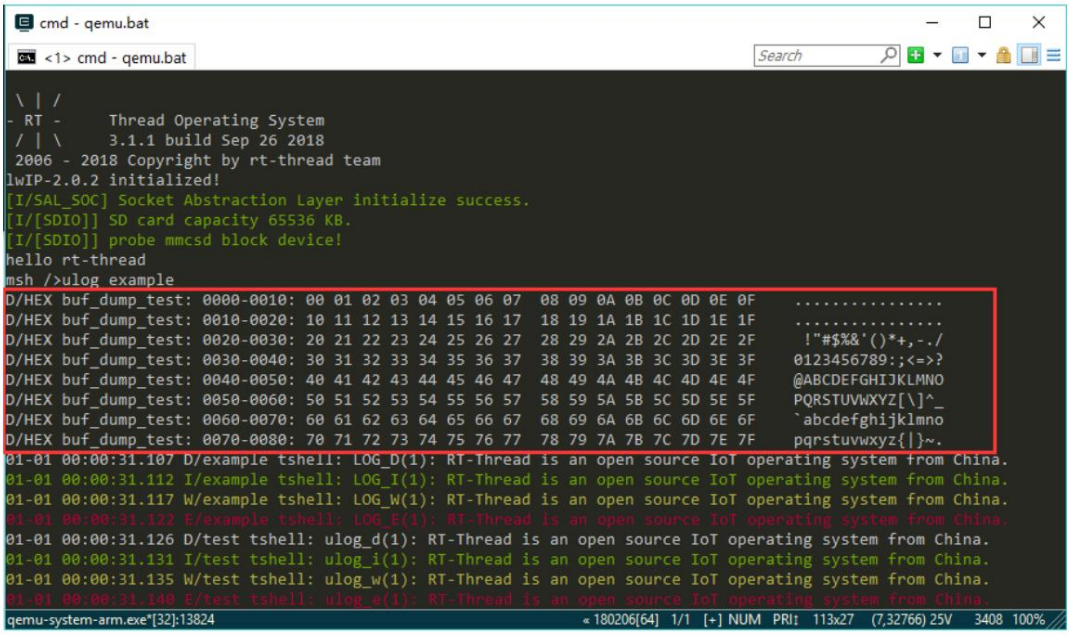

Figure **6:** *ulog* Routines *(hexdump)*

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 Level section to understand the log level classification and check the log filter parameters. Another possibility is that the console backend is accidentally disabled. Just re-enable Enable console backend .

- 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, 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.

5 References

5.1 All relevant **APIs** in this article

5.1.1. **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 also will 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 first capital letter of different levels.

> Note: Before using this API, you need to define LOG_TAG and LOG_LVL in 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