AIROHA

Airoha IoT SDK for RTOS System Log Developer's Guide

Version: 1.3

Release date: 13 January 2017

Document Revision History

Revision	Date	Description
1.0	7 March 2016	Initial release
1.1	2 May 2016	Refined the logging examples with the SDK.
1.2	30 June 2016	Added the log setting commands.
1.3	13 January 2017	Added the exception handler section.

Table of contents

Lists of tables and figures

1.   Introduction

Airoha IoT development platform for RTOS provides comprehensive support to develop and connect Wearables and IoT applications and devices. System Log facilitates debugging during software development and provides a convenient logging system that supports log filtering and multitasking. Log filtering allows the developers to focus on the specific parts of the logs. Multitasking support ensures the logging is properly managed when multiple tasks call the log API, simultaneously. More details about the System Log API usage, can be found in the header file at `<sdk_root>/kernel/service/inc/syslog.h`.

1.1.   System Log architecture

The System Log architecture is shown in Figure 1. The target prints logs using the UART interface. The logs are then displayed on the host side by calling a terminal program such as [TeraTerm](#) and [Putty](#). Installing and setting up TeraTerm is described in "Terminal application setup" and "Serial port settings" sections of the "Airoha IoT SDK for RTOS Get Started Guide" in `<sdk_root>/doc` folder. The target side of Figure 1 is detailed in section 1.4, "Multitasking support". The host side UART configuration is described in section 3, "Host UART Configuration".
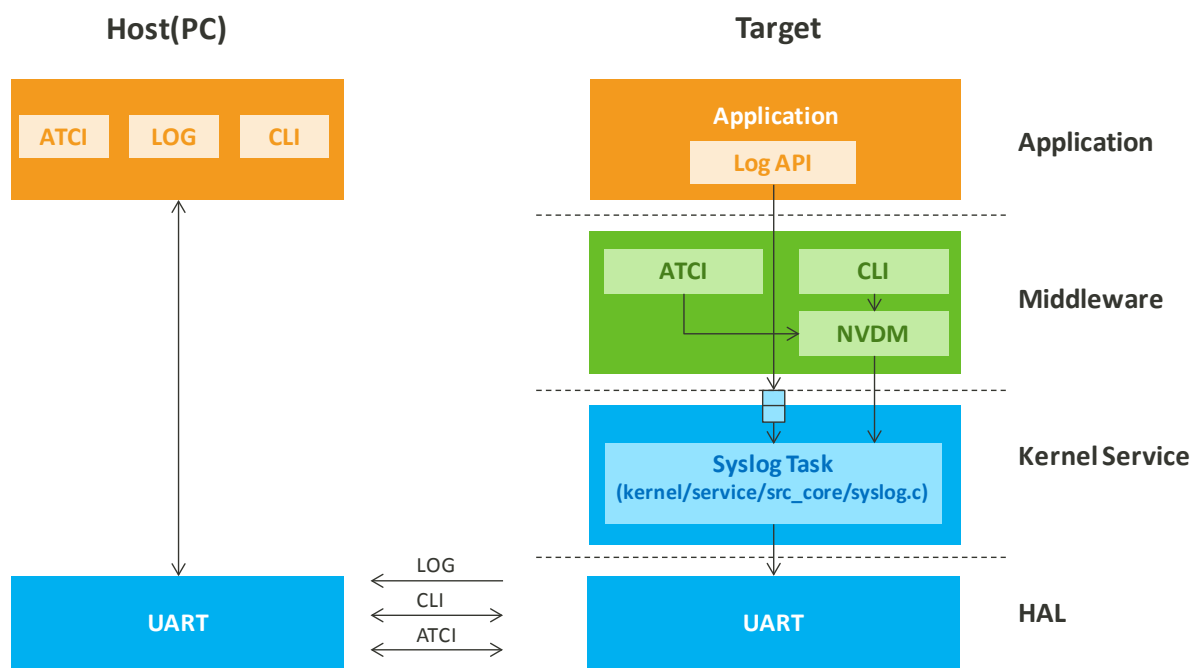
Figure 1. System Log architecture

1.2.   Log filtering

The logs are grouped by modules. Each module is associated with a log control block to specify whether the log messages within that module will be printed or not. The log control block contains the current log level setting for each module. The caller of the log API specifies the log level, the module and the log message. The System Log then performs filtering based on the log level and the current settings in the module's log control block. There are three configurable log levels: INFO, WARNING and ERROR ordered by ascending log level. The log message is printed only if the log level of the current log message is greater than or equal to the log level specified in the log

control block of the module. A common module is defined in `syslog.c` for the type of logs not assigned to any particular module. Users can update the log setting of each module dynamically through the AT commands for LinkIt 2523 HDK or through the CLI commands for LinkIt 7687 HDK. The update takes effect immediately after the commands are issued and log configuration settings are stored in the NVDM.

1.3. Log setting commands

You can create a log module with the function `log_create_module()` (see section 2.1.4, "`log_create_module`"). The system log is enabled by default with the debug level specified in the macro as an input to the function `log_create_module()`.

The log settings are stored in the NVDM. Once the system reboots, the log settings read from the NVDM override the default settings in the image. If there are no log settings in the NVDM, the default log settings in the image are written to the NVDM. The log settings can be updated through AT commands or CLI commands then the updated settings are stored in the NVDM.

The log control blocks are defined in separate modules. For the convenience of log filtering routine implementation, aggregate the log control blocks, see section 1.2, "Log filtering", for all modules of a project into a table and then apply the AT and CLI commands. More details can be found in sections 2.1.1, "`log_init()`", 2.1.2, "LOG_CONTROL_BLOCK_DECLARE" and 2.1.3, "LOG_CONTROL_BLOCK_SYMBOL".

An example description to provide the log setting commands is shown below.

1) Open the main source file of the reference project from `<sdk_root>/project/mt2523_hdk/iot_sdk_demo/src/main.c`.

2) Identify the `LOG_CONTROL_BLOCK_DECLARE(module)` which declares the external reference to the log control block of the given `module`.

```
LOG_CONTROL_BLOCK_DECLARE(atci);
#ifdef MTK_BT_AT_COMMAND_ENABLE
LOG_CONTROL_BLOCK_DECLARE(atci_bt);
#endif
LOG_CONTROL_BLOCK_DECLARE(atci_charger);
LOG_CONTROL_BLOCK_DECLARE(atci_ctp);
LOG_CONTROL_BLOCK_DECLARE(atci_keypad);
LOG_CONTROL_BLOCK_DECLARE(atci_reg);
LOG_CONTROL_BLOCK_DECLARE(atcmd);
LOG_CONTROL_BLOCK_DECLARE(BLE_MESSAGE);
LOG_CONTROL_BLOCK_DECLARE(BLE_STATIC_CB);
LOG_CONTROL_BLOCK_DECLARE(bt);
LOG_CONTROL_BLOCK_DECLARE(bt_audio);
LOG_CONTROL_BLOCK_DECLARE(bt_sink);
LOG_CONTROL_BLOCK_DECLARE(bt_spp);
LOG_CONTROL_BLOCK_DECLARE(common);
LOG_CONTROL_BLOCK_DECLARE(GNSS_TAG);
LOG_CONTROL_BLOCK_DECLARE(GRAPHIC_TAG);
```

```
LOG_CONTROL_BLOCK_DECLARE(hal);

LOG_CONTROL_BLOCK_DECLARE(PXP_MAIN);

LOG_CONTROL_BLOCK_DECLARE(RTC_ATCI);

LOG_CONTROL_BLOCK_DECLARE(sensor);

#ifdef MTK_SMART_BATTERY_ENABLE

LOG_CONTROL_BLOCK_DECLARE(bmt);

#endif
```

3) The table `syslog_control_blocks[]` is a reference to the log control block of each module.

```
log_control_block_t *syslog_control_blocks[] = {

    &LOG_CONTROL_BLOCK_SYMBOL(atci),

#ifdef MTK_BT_AT_COMMAND_ENABLE

    &LOG_CONTROL_BLOCK_SYMBOL(atci_bt),

#endif

    &LOG_CONTROL_BLOCK_SYMBOL(atci_charger),

    &LOG_CONTROL_BLOCK_SYMBOL(atci_ctp),

    &LOG_CONTROL_BLOCK_SYMBOL(atci_keypad),

    &LOG_CONTROL_BLOCK_SYMBOL(atci_reg),

    &LOG_CONTROL_BLOCK_SYMBOL(atcmd),

    &LOG_CONTROL_BLOCK_SYMBOL(BLE_MESSAGE),

    &LOG_CONTROL_BLOCK_SYMBOL(BLE_STATIC_CB),

    &LOG_CONTROL_BLOCK_SYMBOL(bt),

    &LOG_CONTROL_BLOCK_SYMBOL(bt_audio),

    &LOG_CONTROL_BLOCK_SYMBOL(bt_sink),

    &LOG_CONTROL_BLOCK_SYMBOL(bt_spp),

    &LOG_CONTROL_BLOCK_SYMBOL(common),

    &LOG_CONTROL_BLOCK_SYMBOL(GNSS_TAG),

    &LOG_CONTROL_BLOCK_SYMBOL(GRAPHIC_TAG),

    &LOG_CONTROL_BLOCK_SYMBOL(hal),

    &LOG_CONTROL_BLOCK_SYMBOL(PXP_MAIN),

    &LOG_CONTROL_BLOCK_SYMBOL(RTC_ATCI),

    &LOG_CONTROL_BLOCK_SYMBOL(sensor),

#ifdef MTK_SMART_BATTERY_ENABLE

    &LOG_CONTROL_BLOCK_SYMBOL(bmt),

#endif


    NULL
```

```
};
```

4) The user-defined callback functions `syslog_config_save()` and `syslog_config_load()` save and load the log configuration settings of the system.

```
static void syslog_config_save(const syslog_config_t *config)
{
    nvdm_status_t status;
    char *syslog_filter_buf;


    syslog_filter_buf = (char*)pvPortMalloc(SYSLOG_FILTER_LEN);
    configASSERT(syslog_filter_buf != NULL);
    syslog_convert_filter_val2str((const log_control_block_t
**)config->filters, syslog_filter_buf);
    status = nvdm_write_data_item("common",
                                  "syslog_filters",
                                  NVDM_DATA_ITEM_TYPE_STRING,
                                  (const uint8_t *)syslog_filter_buf,
                                  strlen(syslog_filter_buf));
    vPortFree(syslog_filter_buf);
    LOG_I(common, "syslog config save, status=%d", status);
}
static uint32_t syslog_config_load(syslog_config_t *config)
{
    uint32_t sz = SYSLOG_FILTER_LEN;
    char *syslog_filter_buf;


    syslog_filter_buf = (char*)pvPortMalloc(SYSLOG_FILTER_LEN);
    configASSERT(syslog_filter_buf != NULL);
    if (nvdm_read_data_item("common", "syslog_filters",
(uint8_t*)syslog_filter_buf, &sz) == NVDM_STATUS_OK) {
        syslog_convert_filter_str2val(config->filters,
syslog_filter_buf);
    } else {
        /* popuplate the syslog nvdm with the image setting */
        syslog_config_save(config);
    }
    vPortFree(syslog_filter_buf);
    return 0;
```

```
    }
```

5) The log configuration settings are stored in NVDM. Call the function `nvdm_init()` before calling the function `log_init()`. `syslog_config_save()`, `syslog_config_load()` and `syslog_control_blocks[]` are the input parameters for the `log_init()` function.

```c
int main(void)
{
    /* SystemClock Config */
    SystemClock_Config();
    SystemCoreClockUpdate();


    /* init UART earlier */
    log_uart_init(HAL_UART_0);


    /* Configure the hardware ready to run the test. */
    prvSetupHardware();
...


#ifdef NVDM_DEMO
    nvdm_init();
#endif
#if defined(MTK_PORT_SERVICE_ENABLE)
    syslog_port_service_init();
#endif
    log_init(syslog_config_save, syslog_config_load,
syslog_control_blocks);
...


}
```

1.3.1.   AT Commands

The AT commands are applicable to LinkIt 2523 HDK only, as shown in Table 1.

Table 1. AT commands for LinkIt 2523 HDK

Description	Command
Query for the usage	AT+SYSLOG=?
Query for the current setting	AT+SYSLOG?
Update the setting of a module	AT+SYSLOG=<module>,<log_switch>,<print_level>

Description	Command
	`<module>: "module name"`
	`<log_swith>: 0, 1 (0 means on, 1 means off)`
	`<print_level>: 0, 1, 2 (0 – INFO, 1 – WARNING, 2 – ERROR)`

1.3.1.1. AT Command example

In this example, the common and HAL log modules are presented.

1) Execute the command AT+SYSLOG? to query the log settings of the system.

2) From the query result, search for the configuration for HAL (`"hal",0,1`) which indicates the HAL log is enabled and the debug level is set to WARNING to print the HAL warning and error messages.

3) Execute the AT command AT+SYSLOG="hal",0,0 to change the debugging level of HAL to INFO level. You can confirm the update by executing the command AT+SYSLOG?. The HAL setting will become (`"hal",0,0`) after the command is issued and the HAL INFO message will be printed in the log terminal.

The log settings for HAL remain effective even after the device resets.

To update the settings for all modules, use the wild card `"*"`, such as the command AT+SYSLOG="*",1,1 turns off the system logging and changes the debug level to WARNING for all modules.

```
AT+SYSLOG=?

+SYSLOG:

(AT+SYSLOG?, query for the current setting)

(AT+SYSLOG=<module>,<log_switch>,<print_level>, config module's setting)

(<log_switch> = 0|1(meaning on|off), <print_level=0|1|2(meaning I/W/E))

OK

AT+SYSLOG?

+SYSLOG:
("a2dp",1,0),("atci",0,0),("atci_bt",0,0),("atci_charger",0,0),("atci_ctp",0
,0),("atci_keypad",0,0),("atci_reg",0,0),("atcmd",0,0),("audio_middleware_ap
i",0,0),("audio_nvdm",0,0),("av_util",0,0),("avrcp",1,0),("ble_gatt_bqb",0,0
),("ble_gatt_cmd",0,0),("ble_l2cap_bqb",0,0),("ble_sm_bqb",0,0),("BLEBQB",0,
0),("bmt",0,0),("BQBMAIN",0,0),("bt",0,0),("bt_clm",0,0),("bt_pbapc",0,0),("
bt_pbapc_clm",0,0),("bt_sdp_bqb",0,0),("bt_spp_bqb",0,0),("CLM_GAP",0,0),("c
ommon",0,0),("DOGP_ADP",0,0),("fota_dl_m",0,0),("GAP_BQB",0,0),("GATTS_SRV",
0,0),("GNSS_TAG",0,0),("hal",0,1),("RTC_ATCI",0,0),("sensor",0,0),("testfram
ework",0,0),("SPP_PORT",0,1),("bt_spp",0,1)

OK

AT+SYSLOG="hal",0,0

OK

AT+SYSLOG?

+SYSLOG:
("a2dp",1,0),("atci",0,0),("atci_bt",0,0),("atci_charger",0,0),("atci_ctp",0
,0),("atci_keypad",0,0),("atci_reg",0,0),("atcmd",0,0),("audio_middleware_ap
i",0,0),("audio_nvdm",0,0),("av_util",0,0),("avrcp",1,0),("ble_gatt_bqb",0,0
),("ble_gatt_cmd",0,0),("ble_l2cap_bqb",0,0),("ble_sm_bqb",0,0),("BLEBQB",0,
0),("bmt",0,0),("BQBMAIN",0,0),("bt",0,0),("bt_clm",0,0),("bt_pbapc",0,0),("
```

```
bt_pbapc_clm",0,0),("bt_sdp_bqb",0,0),("bt_spp_bqb",0,0),("CLM_GAP",0,0),("c
ommon",0,0),("DOGP_ADP",0,0),("fota_dl_m",0,0),("GAP_BQB",0,0),("GATTS_SRV",
0,0),("GNSS_TAG",0,0),("hal",0,0),("RTC_ATCI",0,0),("sensor",0,0),("testfram
ework",0,0),("SPP_PORT",0,1),("bt_spp",0,1)

OK
```

1.3.2.   CLI commands

The CLI commands are applicable to LinkIt 7687 HDK only, as shown in Table 2.

Table 2. CLI commands for the LinkIt 7687 HDK

Function	Command
Query for the usage	`$log set`
Query for the current setting	`$log`

Update the settings of a module	`$log set <module> <log_switch> <print_level>`
	
	`<module>: module name`
	
	`<log_swith>: on, off`
	
	`<print_level>: info, warning, error`

1.3.2.1.   CLI command example

In this example, the following log modules are presented — main, common, HAL, lwIP, minisupp, inband, Wi-Fi and Bluetooth. The HAL log is enabled and the debug level is set as warning to print warning and error messages.

The debug level of HAL is updated to INFO with the CLI command - `$log set hal on info`. After the command is issued, the HAL INFO message will be printed in the log terminal. The setting for HAL is still effective after the device is powered down and up again. There is a special command by specifying the module name as the wild card "*" to update the setting for all modules. For example, the command `$log set * off warning` will turn off logging and update the debug level as WARNING for all modules.

```
$ log
log
module   on/off   level
------   ------   -----
main     on       error
common   on       warning
hal      on       warning
lwip     on       error
minisupp      on        error
inband   on       warning
wifi     on       error
BT       on       error
```

```
$ log set

log set

required parameters: <module_name> <log_switch> <print_level>

<log_switch>   := on | off

<print_level> := info | warning | error


$ log set hal on info

log set hal on info


$ log

log

module   on/off   level

------   ------   -----

main     on       error

common   on       warning

hal      on       info

lwip     on       error

minisupp          on       error

inband   on       warning

wifi     on       error

BT       on       error
```

1.4. Multitasking support

Since there is only one serial port for system logging, a single log task is created to write the output log data to the serial port when it is scheduled. The other tasks call the System Log API to allocate a log buffer, format the log data by calling the C library function offered by the toolchain and send the log data to the log task. A fixed number of log buffers are allocated as a shared resource. The allocation and release of a log buffer is protected to ensure only one task can access it at a time.

1.5. Supported `printf()` function

Syslog calls the `printf()` function implemented in the toolchain (GCC, KEIL, IAR). In GCC, the floating print function (`%f`) is not enabled by default. To enable it, specify it in linker flag (`LDFLAGS += -u _printf_float`). However, it should be noted that the code size will increase if floating print is enabled.

1.6. Conditional compile options

The project makefile's feature flag MTK_DEBUG_LEVEL defines whether a particular debug log will be compiled. It can be configured in project's `feature.m`k makefile. More compile options are described in Table 3.

Table 3. Compile options for logging

MTK_DEBUG_LEVEL	The effect of the setting

MTK_DEBUG_LEVEL	The effect of the setting
none	All logs are removed.
info	LOG_I, LOG_W, LOG_E, LOG_HEXDUMP_I, LOG_HEXDUMP_W, LOG_HEXDUMP_E are compiled.
warning	LOG_W, LOG_E, LOG_HEXDUMP_W, LOG_HEXDUMP_E are compiled.
error	LOG_E, LOG_HEXDUMP_E are compiled.

2.   System Log APIs

This section describes the source code and header file hierarchy for the System Log and their usage. The files could be found under `<sdk_root>/kernel/service/src_core.`

Table 4 provides details on the main functions of the System Log tree hierarchy.

Table 4. System Log source code description

File	Description
`<project>/GCC/feature.mk`	Define `MTK_DEBUG_LEVEL`
`kernel/service/inc/syslog.h`	The main header file for the System Log. It includes the macros and APIs for initialization, logging and filtering purposes.
`kernel/service/src/src_core/syslog.c`	Contains the internal implementation of the System Log API. It also declares the log control block for a common module that enables the user to print log messages of the common module.
`kernel/service/src/syslog_cli.c`	Includes the syslog CLI commands for LinkIt 7687 HDK.
`middleware/MTK/atci/at_command/ at_command_syslog.c`	Includes the syslog AT commands for LinkIt 2523 HDK.

2.1.   Supported APIs

The following lists the most commonly used APIs.

2.1.1.   `log_init()`

Description	This function initializes the system log. The syslog control blocks table (see section 1.3, "Log setting commands") is used in syslog related AT and CLI commands.
Parameter	**syslog_save_fn save**. User-defined callback function to store log settings. **syslog_load_fn load**. User-defined callback function to read log settings. **log_control_block_t **entries**. A pointer to syslog control blocks table for the project.

2.1.2.   `LOG_CONTROL_BLOCK_DECLARE()`

Description	This macro declares the external reference to a log control block (see section 1.3, "Log setting commands") of a module.
Parameter	**module:** Module name.

2.1.3.   `LOG_CONTROL_BLOCK_SYMBOL`

Description	This macro refers to the log control block (see section 1.3, "Log setting commands") of a module.
Parameter	**module**. Module name.

2.1.4.   `log_create_module()`

Description	This macro creates the log control block of a module.

Parameter	**module**. Module name.
	level. "PRINT_LEVEL_INFO", the debug level for the module is INFO.
	"PRINT_LEVEL_WARNING", the debug level for the module is WARNING.
	"PRINT_LEVEL_ERROR", the debug level for the module is ERROR.

2.1.5.   log_config_print_switch()

Description	This macro configures whether the log for the module is enabled.
Parameter	**module**. Module name.
	log_switch. "DEBUG_LOG_ON", the log for the module is enabled.
	"DEBUG_LOG_OFF", the log for the module is disabled.

2.1.6.   log_config_print_level()

Description	This macro configures the debug level for the module.
Parameter	**module**. Module name.
	log_switch. "PRINT_LEVEL_INFO", the debug level of the module is INFO.
	"PRINT_LEVEL_WARNING", the log level of the module is WARNING.
	"PRINT_LEVEL_ERROR", the debug level of the module is ERROR.

2.1.7.   LOG_I

Description	This macro adds an INFO log message.
Parameter	**module**. Module name.
	message. Format specifiers.
	variadic arguments. The parameters corresponding to the format specifiers defined in the **message** parameter.

2.1.8.   LOG_W

Description	This macro adds a WARNING log message.
Parameter	**module**. Module name.
	message. Format specifiers.
	variadic arguments. The parameters corresponding to the format specifiers defined in the **message** parameter.

2.1.9.   LOG_E

Description	This macro adds an ERROR log message.
Parameter	**module**. The module name.
	message. Format specifiers.
	variadic arguments. The parameters corresponding to the format specifiers defined in the **message** parameter.

2.1.10.   LOG_HEXDUMP_I

Description	This macro adds an INFO log message and displays the contents of a specified range of

	memory. The memory is displayed in hexadecimal format.
Parameter	**module**. The module name.
	message. Format specifiers.
	data. The start address of the memory region to be displayed.
	len. The length of the memory region to be displayed, in bytes.
	variadic arguments. The parameters corresponding to the format specifiers defined in the **message** parameter.

2.1.11.   LOG_HEXDUMP_W

Description	This macro adds a WARNING log message and displays the contents of a specified range of memory. The memory is displayed in hexadecimal format.
Parameter	**module**. The module name.
	message. Format specifiers.
	data. The start address of the memory region to be displayed.
	len. The length of the memory region to be displayed, in bytes.
	variadic arguments. The parameters corresponding to the format specifiers defined in the **message** parameter.

2.1.12.   LOG_HEXDUMP_E

Description	This macro adds an ERROR log message and displays the contents of a specified range of memory. The memory is displayed in hexadecimal format.
Parameter	**module**. The module name.
	message. Format specifiers.
	data. The start address of the memory region to be displayed.
	len. The length of the memory region to be displayed, in bytes.
	variadic arguments. The parameters corresponding to the format specifiers defined in the **message** parameter.

2.2.   System Log API usage

In this section, the System Log APIs are utilized to create and configure logs for a specific module. A user defined function is then invoked to demonstrate the logging operation for different levels of log configuration.

The code below declares a log control block for the Hardware Abstraction Layer (HAL) module. In this example, the log level for HAL is configured as PRINT_LEVEL_WARNING. Call `log_create_module` to set the configuration, as shown below.

```
#include "syslog.h"
log_create_module(hal, PRINT_LEVEL_WARNING);
```

A log control block for the HAL module with the content shown below is created:

```
log_control_block_t log_control_block_hal =
{
    .module_name = "hal",
    .log_switch = (DEBUG_LOG_ON),
```

```
    .print_level = ((PRINT_LEVEL_WARNING)),

    .f_print_handler = default_print_ext,

    .f_dump_buffer = default_dump_ext
};
```

The following user defined function (`demo_function()`) demonstrates the usage of the System Log API. Only the log messages with a print level greater than or equal to PRINT_LEVEL_WARNING will be printed.

```
void demo_function(int id1, int id2, char *statement)
{
    LOG_I(hal, "INFO message %d\n", id1);        /* It is not printed */

    LOG_W(hal, "WARNING message %d\n", id2);     /* It is printed */

    LOG_E(hal, "ERROR message %s\n", statement); /* It is printed */
}
```

2.3. System Log output format

Each log message includes a log header and a log body. The log header shows **timestamp**, **module name**, **level**, **function**, and **line number**. The log body shows the user-defined log.

```
[T: 20761 M: hal C: INFO F: vTestTask L: 824]: CM4 Task 0 Hello World, Idx =
10
```

2.4. System Logs attached to a task

System Logs could be associated with hardware related tasks. Apply the reference application (`freertos_thread_creation`) available in the SDK to create your own applications. System Log APIs are called to print the last execution time and the stack dump of a specific task.

The reference application can be found at `<sdk_root>/project/mt2523_hdk/apps/freertos_thread_creation/src/main.c` and `<sdk_root>/project/mt7687_hdk/apps/freertos_thread_creation/src/main.c`.

There are four instances of `vTestTask` task created. Each instance of the `vTestTask` task prints its own logs independently.

    1) Creating the tasks.

Create four instances of `vTestTask` in main function by calling `xTaskCreate()` function. Each task requires a task number. The `idx` parameter represents the task number.

```
//Create the log control block for freertos module

log_create_module(hal, PRINT_LEVEL_INFO);


int main(void)
 {
```

```
...
   int idx=0;


...
      for(idx=0;idx<4;idx++)
      {
       xTaskCreate(vTestTask, "Test", 1024, (void*)idx,2, NULL);


      }
...
  }
```

2) Declaring and initializing the variables.

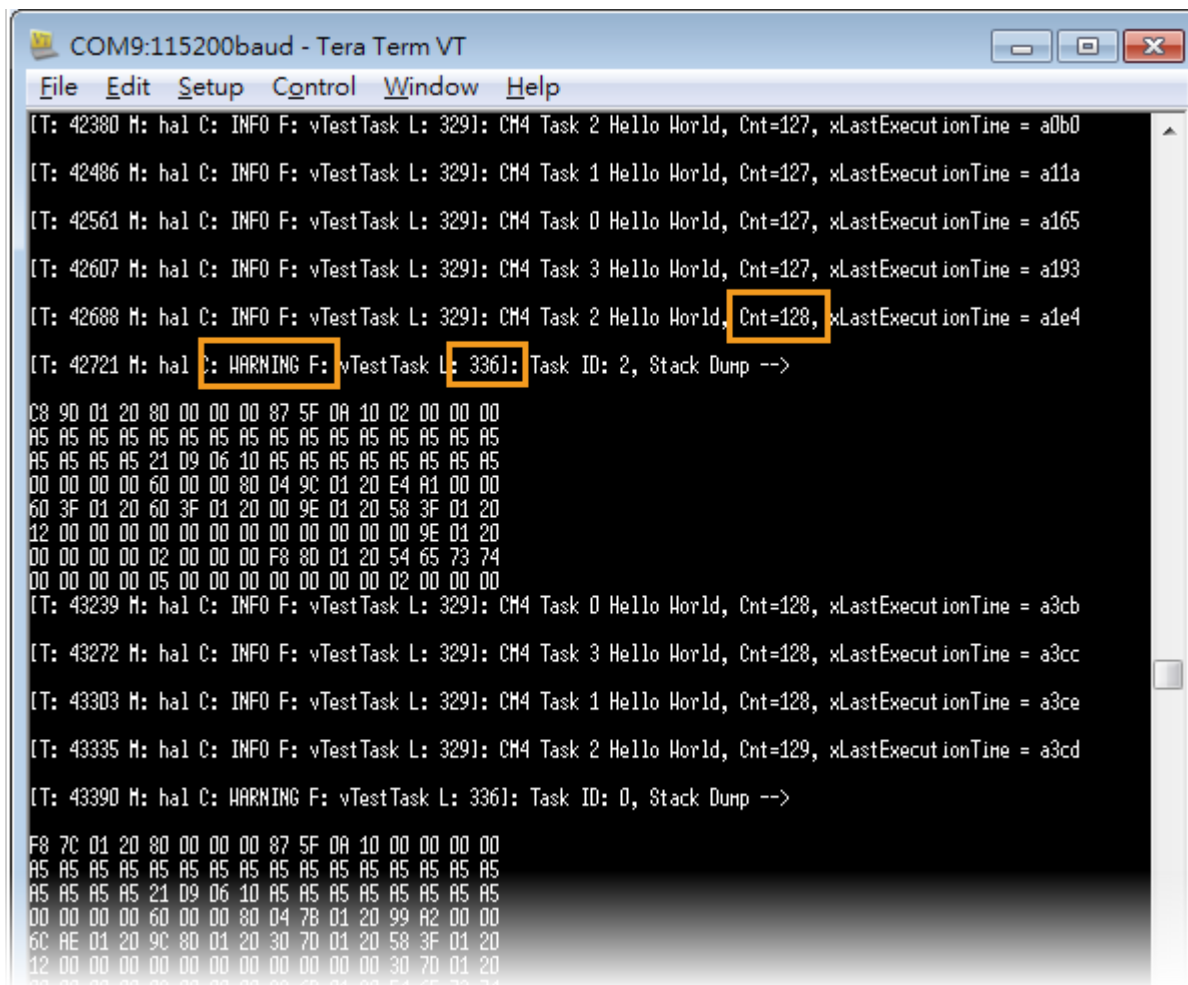The function begins with declaring several parameters for further use, as follows:

```
static void vTestTask( void *pvParameters )
 {
     int i=0;
     int idx = (int)pvParameters;
     portTickType xLastExecutionTime;
     char * stack;
```

3) Continuous looping.

Start a continuous `while` loop, then call a function `vTaskDelay(300)` to repeat the operation of the loop content for each 300 milliseconds. The loop index `i` is then updated and the variable `xLastExecutionTime` is assigned to the system tick count obtained from `xTaskGetTickCount()` function.

```
     while (1) {
         vTaskDelay(300);
         i++;
         //configASSERT(i < 1000);
         xLastExecutionTime = xTaskGetTickCount();
```

4) Printing the System Logs.

In line 329 of `vTestTask()` function, the INFO log message is printed in every loop.

In line 333 of `vTestTask()` function, the INFO log message with memory dump is printed, if the loop index `i` is 256*n, where n is a positive integer (≥0).

In line 336 of `vTestTask()` function, the WARNING log message with memory dump is printed, if the loop index `i` is (256*n + 128), where n is a positive integer (≥0).

```
LOG_I (hal, "CM4 Task %d Hello World, Cnt=%d, xLastExecutionTime = %x\n",
idx, i, (int)xLastExecutionTime);
```

```
        if (i % 128 == 0) {
            __asm volatile ("mov %0, r13\n" : "=r" (stack) );
            if ( i % 256 == 0) {
                LOG_HEXDUMP_I (hal, "Task ID: %d, Stack Dump -->\n",
(char*)stack, 128, idx);
            }
            else {
                LOG_HEXDUMP_W (hal, "Task ID: %d, Stack Dump -->\n",
(char*)stack, 128, idx);
            }
```
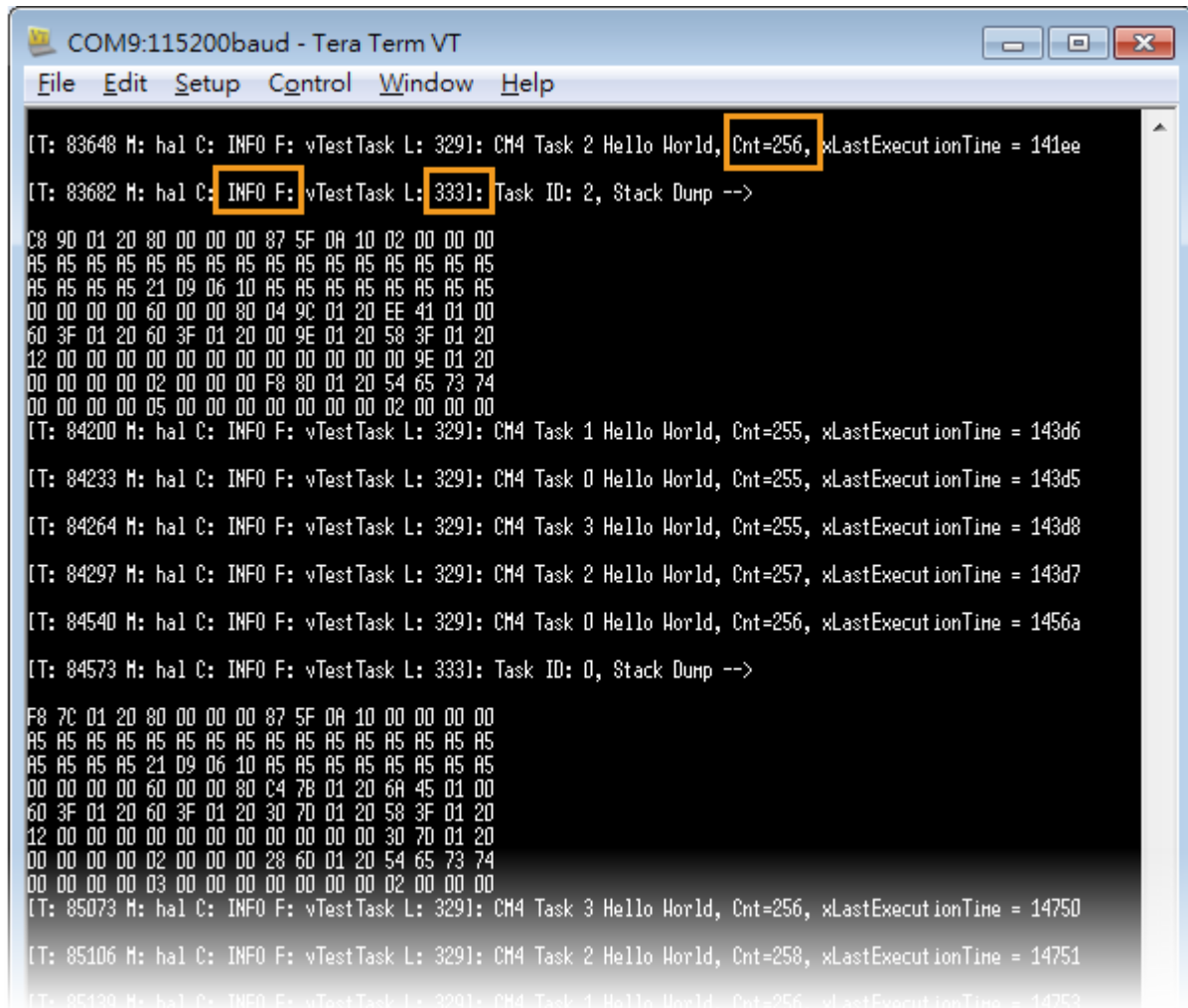
The full source code of the function is shown below:

```
static void vTestTask( void *pvParameters )
{
    int i=0;
    int idx = (int)pvParameters;
    portTickType xLastExecutionTime;
    char * stack;

    while (1) {
        vTaskDelay(300);
        i++;
        //configASSERT(i < 1000);
        xLastExecutionTime = xTaskGetTickCount();
        LOG_I(hal, "CM4 Task %d Hello World, Cnt=%d, xLastExecutionTime =
%x\n", idx, i, (int)xLastExecutionTime);
        if (i % 128 == 0) {
            __asm volatile ("mov %0, r13\n" : "=r" (stack) );
            if ( i % 256 == 0) {
                LOG_HEXDUMP_I (hal, "Task ID: %d, Stack Dump -->\n",
(char*)stack, 128, idx);
            }
            else {
                LOG_HEXDUMP_W (hal, "Task ID: %d, Stack Dump -->\n",
(char*)stack, 128, idx);
            }
        }
    }
}
```

A snapshot of execution result is shown in Figure 2 and Figure 3 with a loop index of 128 and 256, respectively.

Figure 2. Example output of `freertos_thread_creation` *(loop index = 128)*

Figure 3. Example output of `freertos_thread_creation` *(loop index = 256)*

3. Host UART Configuration

The System Log messages on the host side are displayed using a terminal program, such TeraTerm and Putty. The terminal program is able to receive text data from the serial port. TeraTerm terminal program is selected for the demonstration purposes. The hardware development board is the LinkIt 7687 development board.

The host UART settings are configured as follows:

1) Set up the port.

   a) Open Windows Control Panel then click **System** and:

      o On Windows 7 and 8, click **Device Manager**.

      o On Windows XP, click the **Hardware** tab and then **Device Manager**.

   b) In **Device Manager**, navigate to **Ports (COM & LPT)** (see Figure 4).

   c) Connect the development board to your computer using a Micro-USB cable.

   d) A new **COM** device should appear under **Ports (COM & LPT)** in **Device Manager**, as shown in Figure 4. Note the **COMx** port number of the serial communication port, this information is needed to complete configuration of the Tera Term terminal program.

2) Launch the Tera Term terminal program and open the **Serial port setup** window.

   a) Assign the COM port number found in Set up the port. The rest of the parameters such as **Baud rate**, **Data**, **Parity**, **Stop** and **Flow control** should be defined, as shown in Figure 5.
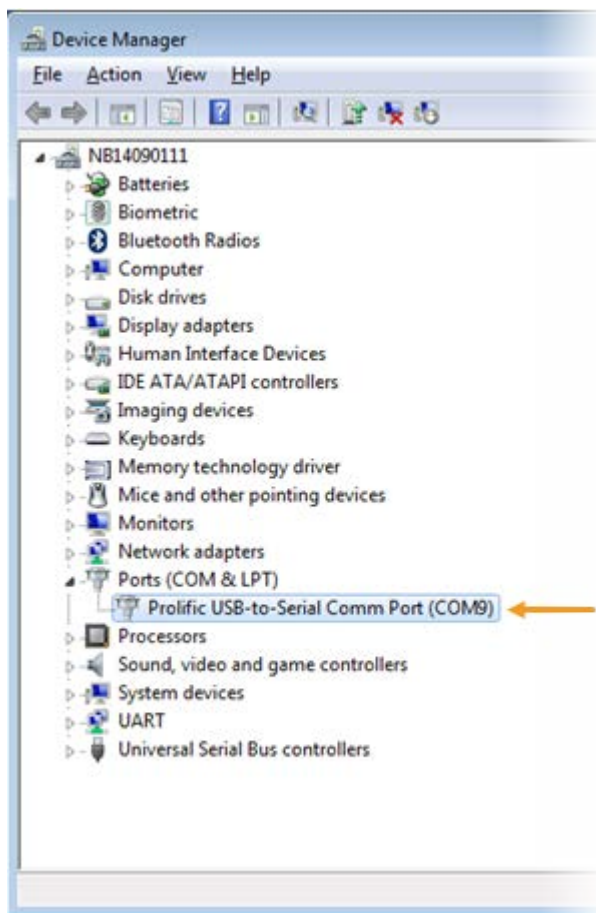
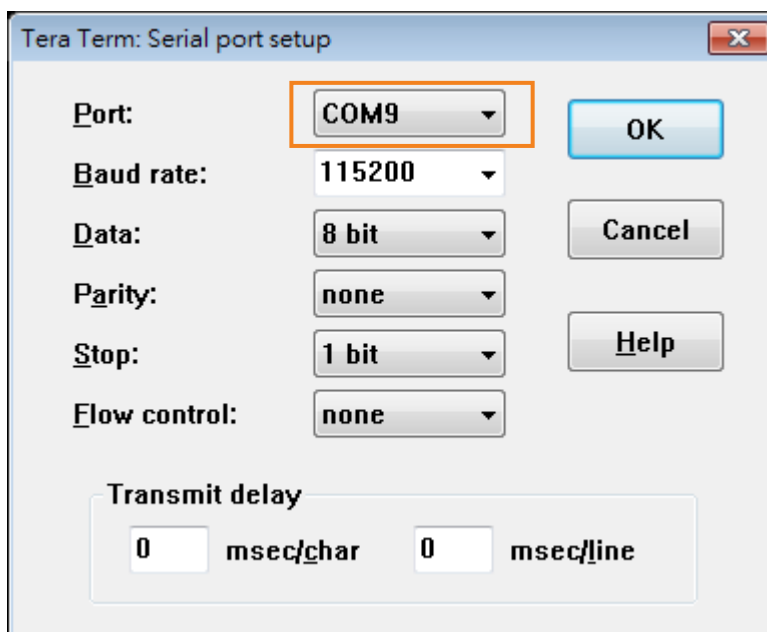*Figure 4. Enumerated COM port on the host*

Figure 5. UART configurations in Tera Term terminal program

4. Exception Handler

Exception handler assists in debugging programming errors. When exception occurs, the content of the processor's core registers and memory are dumped into a log file. The data in the dump file can be analyzed given the symbol information in the corresponding ELF file. Exceptions are either processor detected exceptions or user defined assert failures. Four types of hardware exceptions are supported in Exception Handler for ARM Cortex-M4: **HardFault**, **MemMange**, **BusFault**, and **UsageFault**. Refer to ARM v7-M Architecture Reference Manual for more detail.

Table 5 describes the source code and header file hierarchy for the Exception Handler and their usage.

Table 5 Exception Handler source code description

File	Description
`kernel/service/inc/exception_handler.h`	The main header file for the Exception Handler. It includes data types and APIs.
`kernel/service/src/src/memory_regions.c`	Contains the system memory map definition to facilitate the Exception Handler to dump memory. It is mainly a table of linker generated symbols of the system.
`kernel/service/src_core/exception_handler.c`	Contains the internal implementation of the Exception Handler.

4.1. Supported APIs

The following lists the most commonly used APIs.

4.1.1. configASSERT()

Description	Use this macro to add assert expression. If the expression is evaluated as FALSE, `platform_assert()` is called.
Parameter	**expr**. The expression to be evaluated.

4.1.2. abort()

Description	A porting function used in GCC toolchain. It's linked with toolchain's `assert()`.
Parameter	**none**

4.1.3. __aeabi_assert()

Description	A porting function used in Keil or IAR embedded workbench toolchain. It's linked with tool chain's `assert()`.
Parameter	**const char *expr**. User-defined assert expression. **const char *file**. Source file where assert expression is added. **int line**. Line number of the assert expression.

4.1.4.　`platform_assert()`

Description	This function is called when assert check fails. It's used in the `configASSERT` macro. It intentionally performs invalid memory access to bring the system into exception.
Parameter	**const char *expr**. User-defined assert expression. **const char *file**. Source file where assert expression is added. **int line**. Line number of the assert expression.

4.1.5.　`exception_register_callbacks()`

Description	Users can register `init_cb` and or `dump_cb`. `init_cb` is called when exception occurs. It's designed for user-defined functions, such as trigger system log buffer flush operation. `dump_cb` is called after memory dump is complete.
Parameter	**exception_config_type *cb**. User-defined callback functions.

4.1.6.　`exception_dump_config()`

Description	This function is used to customize the exception handler behavior.
Parameter	**int flag**. DISABLE_MEMDUMP_MAGIC: Exception handler calls `exception_reboot()` after dumping the core registers. Other values: no effect, reserved for future use.

4.1.7.　`exception_reboot()`

Description	The exception handler calls this function after the core register is dumped if the memory dump feature is disabled (see section 4.1.6, "`exception_dump_config()`"). `exception_reboot()` is implemented as a weak function in exception handler to allow user customization as the reboot function is chip and project dependent (see section 4.3, "Reboot without memory dump on exception").
Parameter	**none**

4.2.　Enable exception handling in projects

For GCC environment, users need to include `kernel/service/module.mk` in the project's Makefile.

```
# kernel service files

include $(SOURCE_DIR)/kernel/service/module.mk

"project/mt2523_hdk/apps/iot_sdk_demo/GCC/Makefile" 300 lines --25%--
```

To enable exception handling for applications in Keil IDE or IAR embedded workbench environments, include a prebuilt kernel service library and `memory_regions.c` source file into the project's configuration file. An example project in Keil IDE and IAR embedded workbench IDE is shown in Figure 6 Figure 7, respectively.
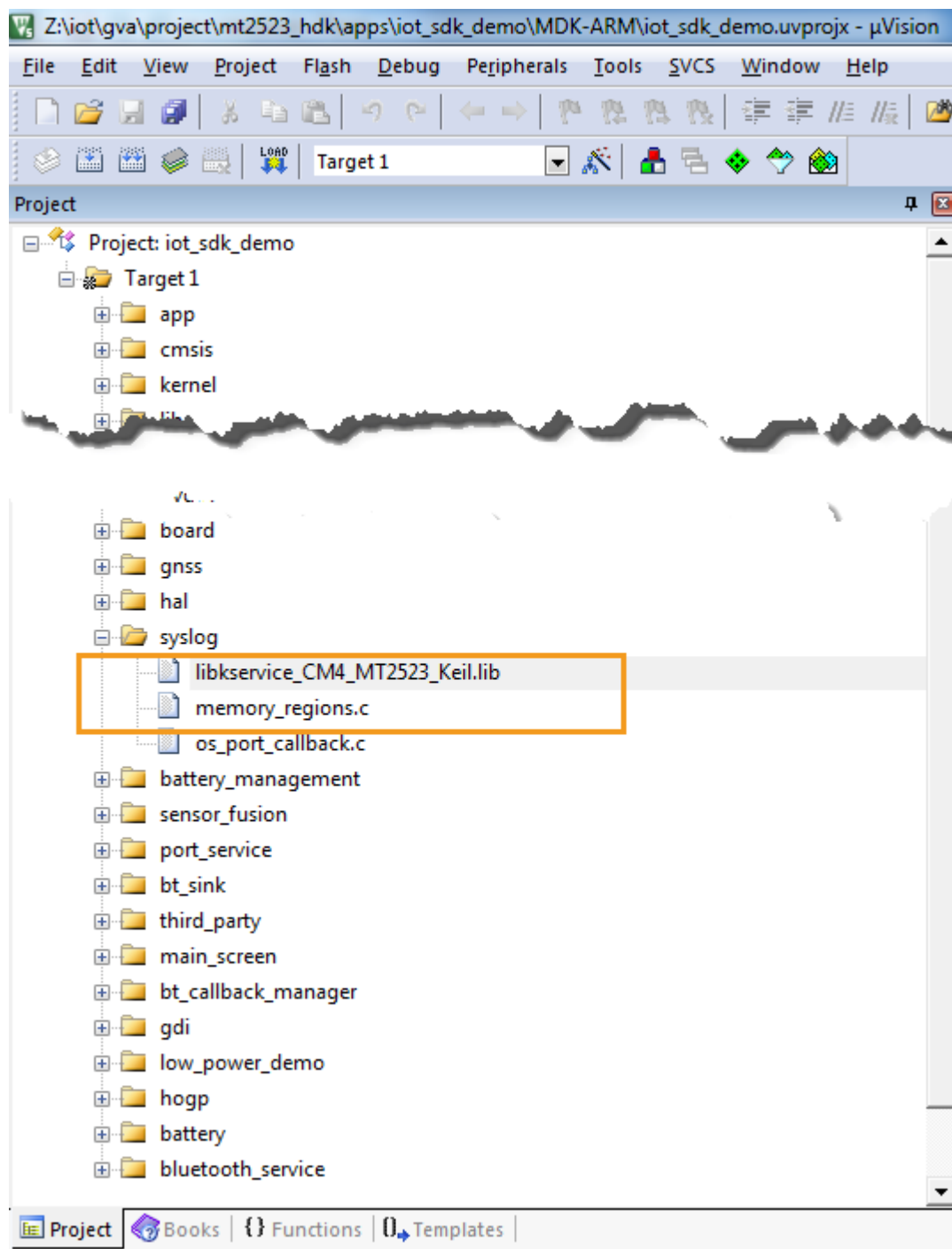
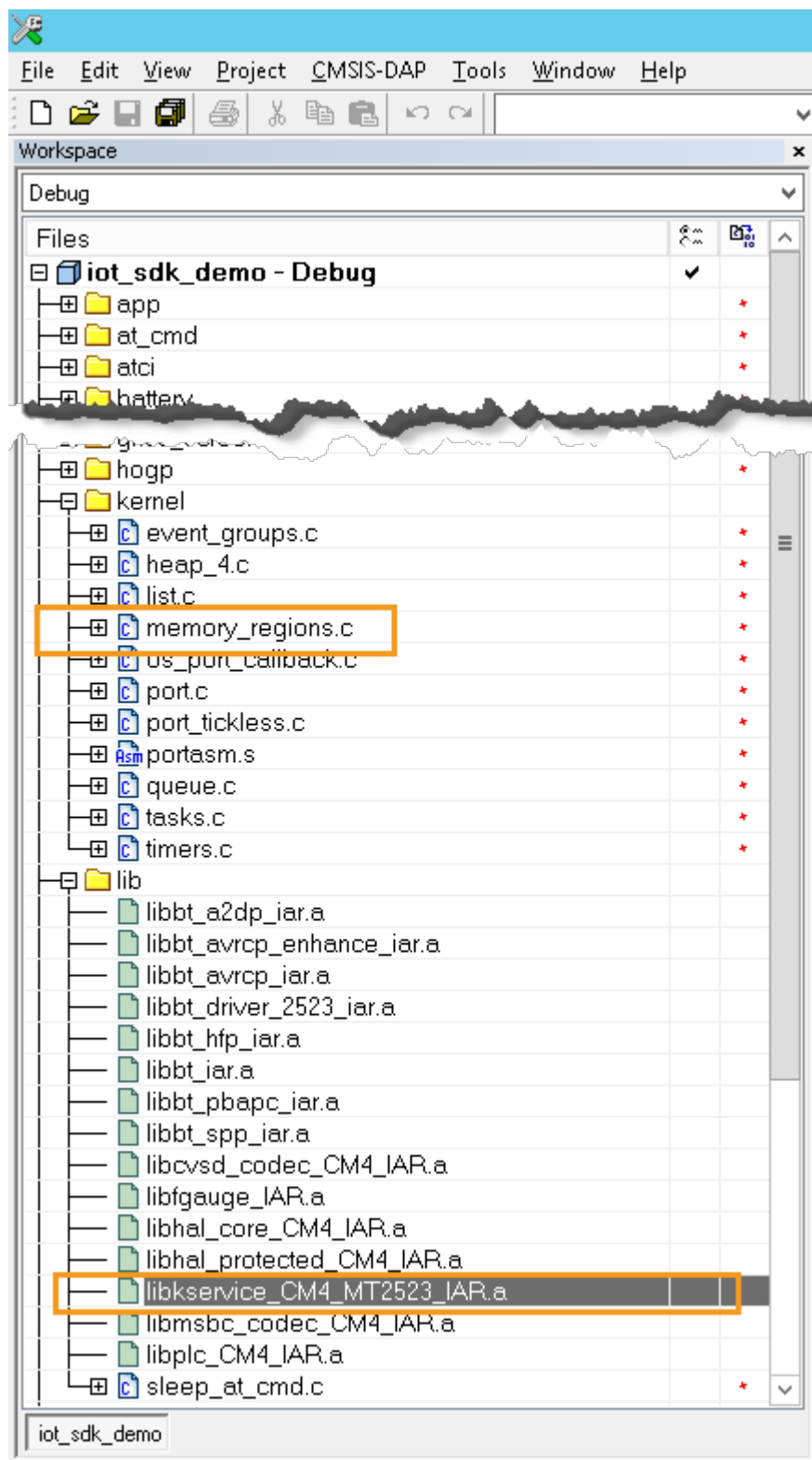*Figure 6. Enable exception handling in Keil IDE*

Figure 7. Enable exception handling in IAR embedded workbench IDE

It's suggested to use `configASSERT()` instead of `assert()`. The `"__noreturn__"` attribute specified for the `assert()` function, informs the compiler that the function does not return. The compiler, performing optimization, may not save the key registers necessary for callstack unwind. The `configASSERT()` macro is defined in the project's `FreeRTOSConfig.h`.

```
#if defined(__ICCARM__) || defined(__CC_ARM) || defined(__GNUC__)

extern void abort(void);

extern void platform_assert(const char *, const char *, int);

...

#define configASSERT( x ) if( (x) == 0 ) { platform_assert(#x, __FILE__,
__LINE__); }

#include "syslog.h"

#endif /*#if defined(__ICCARM__) || defined(__CC_ARM) || defined(__GNUC__)*/

"project/mt2523_hdk/apps/iot_sdk_demo/inc/FreeRTOSConfig.h"
```

The project's memory map defined in `kernel/service/src/memory_regions.c` facilitates the exception handler to create a memory dump. It's a table of the linker-generated symbols specifying the location of image sections.

```
const memory_region_type memory_regions[] =
{
    {"ram_text", Image$$RAM_TEXT$$Base, Image$$RAM_TEXT$$Limit, 1},

    {"noncached_data", Image$$NONCACHED_DATA$$Base,
Image$$NONCACHED_ZI$$Limit, 1},

    {"cached_data", Image$$CACHED_DATA$$RW$$Base,
Image$$CACHED_DATA$$ZI$$Limit, 1},

    {"tcm", Image$$TCM$$RO$$Base, Image$$STACK$$ZI$$Limit, 1},

    {"stack", Image$$STACK$$ZI$$Base, Image$$STACK$$ZI$$Limit, 0},

    {"scs", (unsigned int*)SCS_BASE, (unsigned int*)(SCS_BASE + 0x1000), 1},

    {0}
};

"kernel/service/src/memory_regions.c"
```

4.3. Reboot without memory dump on exception

Users should call `exception_dump_config(DISABLE_MEMDUMP_MAGIC)` during system initialization and implement the `exception_reboot()` function. Normally reboot can be implemented by hardware watchdog. Refer to HAL/WDT section in `<SDK_root>/doc/Airoha IoT SDK for Chipset API Reference Manual.html`, for more details.

4.4. Exception Dump Example

An exception dump examples for MT2523 are shown in Figure 8 to Figure 10. The exception is triggered by the AT command `"AT+SYSTEM=crash"`. When the program reaches the line 244 of `at_command_system.c`, the system crashes due to assert check failure.

```
/* AT command handler  */
atci_status_t atci_cmd_hdlr_system(atci_parse_cmd_param_t *parse_cmd)
{
    char *param = NULL, *saveptr = NULL;
```

```
    atci_response_t *presponse = pvPortMalloc(sizeof(atci_response_t));
...
        case ATCI_CMD_MODE_EXECUTION: /* rec: AT+SYSTEM=<module> */
...
            } else if (strcmp(param, "crash") == 0) {
                strncpy((char *)(presponse->response_buf),
                        "+SYSTEM:\r\n system will crash...\r\n",
                        (int32_t)ATCI_UART_TX_FIFO_BUFFER_SIZE);
                presponse->response_len = strlen((char *)(presponse-
>response_buf));
                presponse->response_flag = ATCI_RESPONSE_FLAG_APPEND_OK;
                atci_send_response(presponse);
                configASSERT(0);
            } else {
...
```

"**middleware/MTK/atci/at_command/at_command_system.c**"

The system is working properly before the AT command is issued which is obvious from the AT command logs.

After an exception occurs, the exception handler initializes and calls the registered exception callbacks. By design, the system log service module registers an exception initialization callback to flush the last minute logs still in the log buffer. There is a hint message "`>>> dump syslog buffer`" followed by the flushed logs ordered by logging time (see Figure 8).
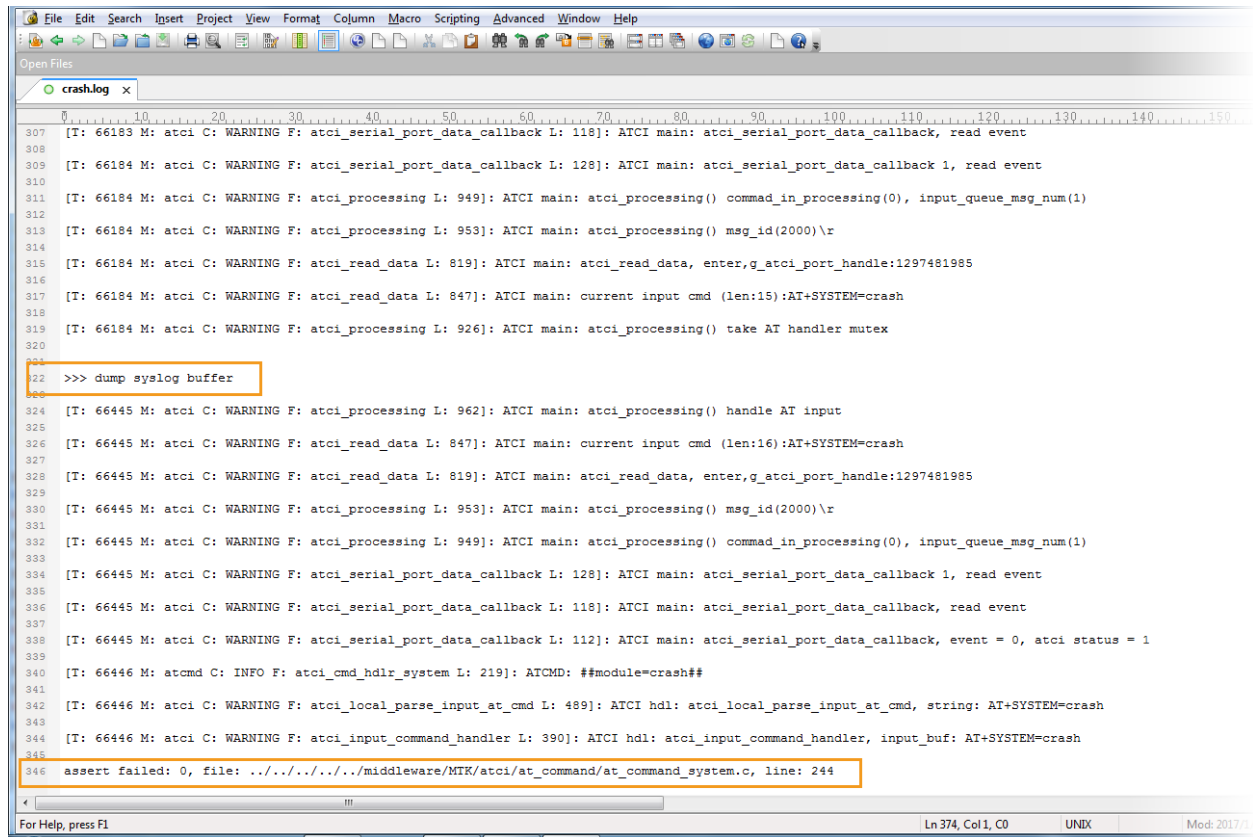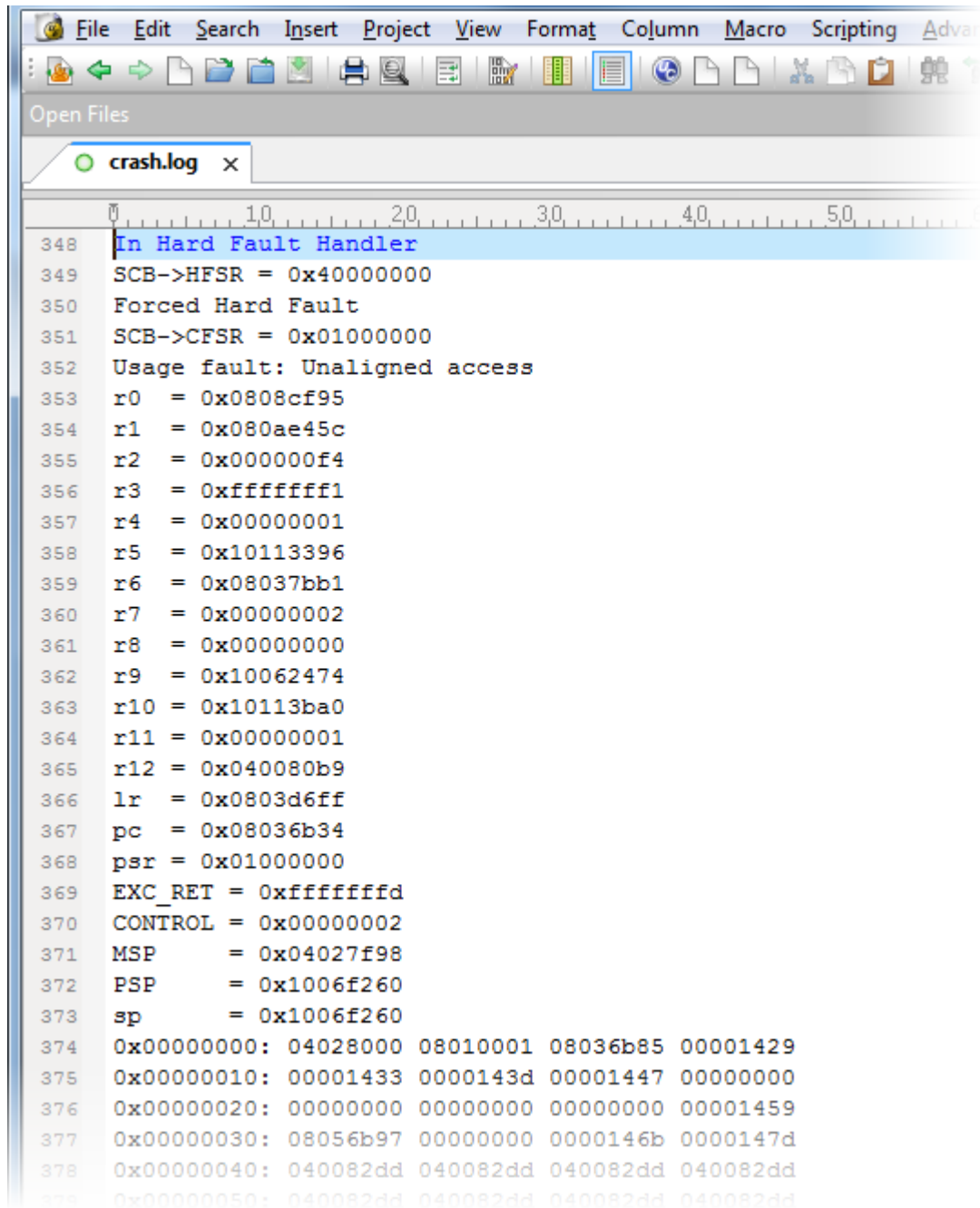
Figure 8. Exception dump Example (1/3)

The asserted expression, source file and line number are displayed next. In this case, the exception occurs in line 244 of `at_command_system.c`.

```
assert failed: 0,
file: ../../../../../middleware/MTK/atci/at_command/at_command_system.c,
line: 244
```

After the "`assert failed:`" message, the exception type and the core registers' values are logged, as shown in Figure 9.

File   Edit   Search   Insert   Project   View   Format   Column   Macro   Scripting   Adva

Open Files

○ crash.log   ×

```
348  In Hard Fault Handler
349  SCB->HFSR = 0x40000000
350  Forced Hard Fault
351  SCB->CFSR = 0x01000000
352  Usage fault: Unaligned access
353  r0  = 0x0808cf95
354  r1  = 0x080ae45c
355  r2  = 0x000000f4
356  r3  = 0xfffffff1
357  r4  = 0x00000001
358  r5  = 0x10113396
359  r6  = 0x08037bb1
360  r7  = 0x00000002
361  r8  = 0x00000000
362  r9  = 0x10062474
363  r10 = 0x10113ba0
364  r11 = 0x00000001
365  r12 = 0x040080b9
366  lr  = 0x0803d6ff
367  pc  = 0x08036b34
368  psr = 0x01000000
369  EXC_RET = 0xfffffffd
370  CONTROL = 0x00000002
371  MSP     = 0x04027f98
372  PSP     = 0x1006f260
373  sp      = 0x1006f260
374  0x00000000: 04028000 08010001 08036b85 00001429
375  0x00000010: 00001433 0000143d 00001447 00000000
376  0x00000020: 00000000 00000000 00000000 00001459
377  0x00000030: 08056b97 00000000 0000146b 0000147d
378  0x00000040: 040082dd 040082dd 040082dd 040082dd
379  0x00000050: 040082dd 040082dd 040082dd 040082dd
```

Figure 9.Exception dump example (2/3)

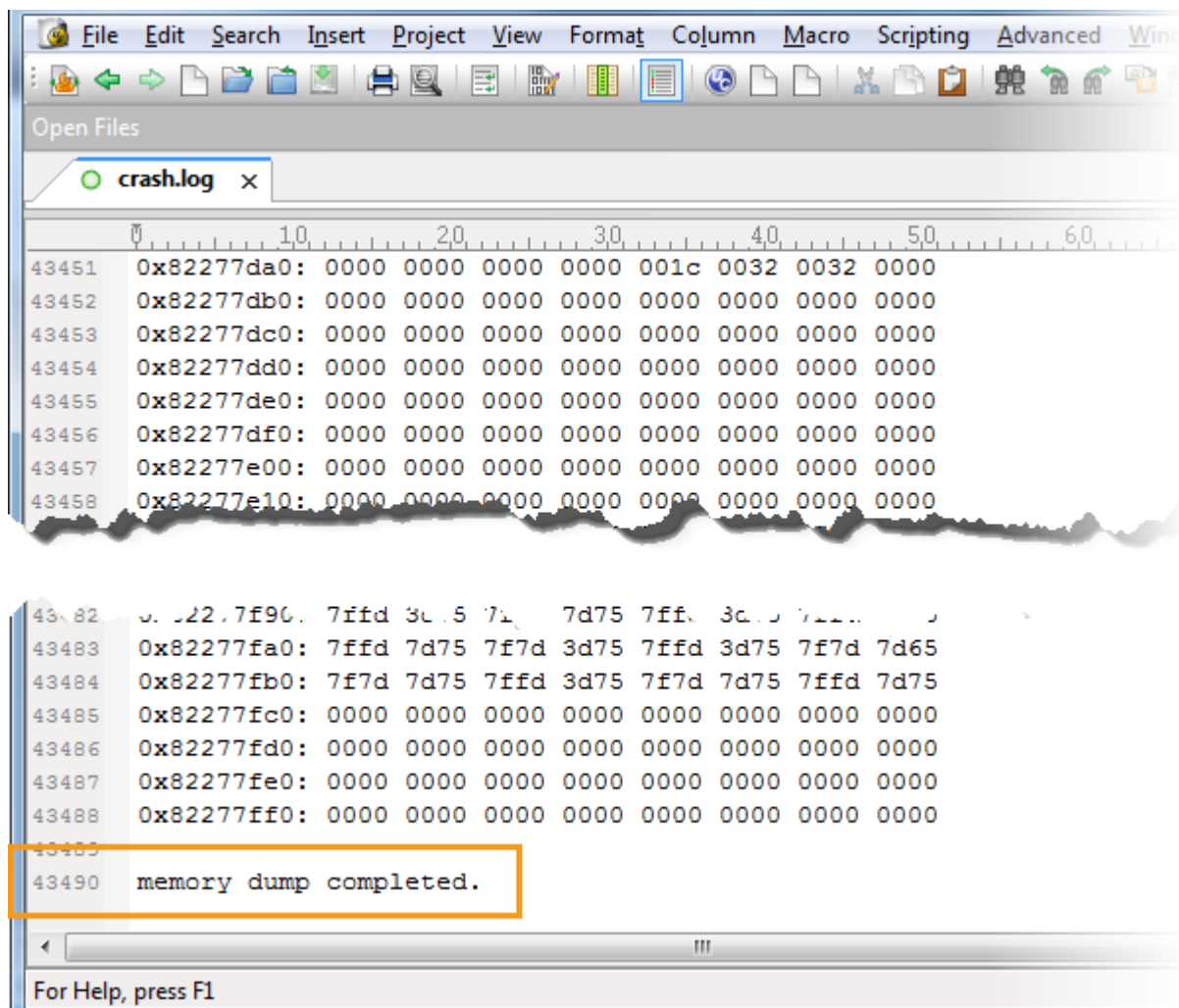Then the system memory is dumped and "`memory dump completed.`" message is displayed, as shown in Figure 10.

Figure 10. Exception dump example (3/3)