Xradio SDK 内存调试工具

使用指南

版本号：1.0

发布时间：2022-01-07

版本历史

版本	日期	责任人	版本描述
1.0	2022-01-07	AWA 1096	创建文档

目录

前言

1.1 文档简介

本文档介绍了 Xradio SDK 的内存调试方法。

1.2 目标读者

使用 Xradio SDK 的开发人员。

1.3 适用范围

本文档适用于 XR872/XR808/XR806 SDK，支持 Xradio 系列芯片。

1.4 文档约定

1.4.1 标志说明

本文档采用各种醒目的标志来表示在操作过程中应该特别注意的地方，这些标志的含义如下：

标识	说明
警告	该标志后的说明应给予格外关注，如果不遵守，可能会导致人员受伤或死亡。
注意	提醒操作中应注意的事项。不当的操作可能会损坏器件，影响可靠性、降低性能等。
说明	为准确理解文中指令、正确实施操作而提供的补充或强调信息。
窍门	一些容易忽视的小功能、技巧。了解这些功能或技巧能帮助解决特定问题或者节省操作时间。

1.4.2 地址与数据描述方法约定

本文档在描述地址、数据时遵循如下约定：

符号	例子	说明
0x	0x0200，0x79	地址或数据以 16 进制表示。
0b	0b010，0b00 000 111	数据采用二进制表示(寄存器描述除外)。
X	00X，XX1	数据描述中，X 代表 0 或 1。 例如，00X 代表 000 或 001；XX1 代表 001，011，101 或 111。

1.4.3 数值单位约定

本文档在描述数据容量（如 NAND 容量）时，单位词头代表的是 1024 的倍数；描述频率、数据速率等时则代表的是 1000 的倍数。具体如下：

类型	符号	对应数值
数据容量（如 NAND 容量）	1 K	1024
	1 M	1 048 576
	1 G	1 073 741 824
频率，数据速率等	1 k	1000
	1 M	1 000 000
	1 G	1 000 000 000

2 概述

嵌入式开发中，小系统的内存空间有限，各种内存问题的排查十分耗时。为了提高排查内存问题的效率，Xradio SDK 提供了堆（heap）使用统计、内存泄漏检查、内存越界检查等若干手段。

3 堆统计

3.1 粗略堆统计

Xradio SDK 的堆管理采用了标准 C 库中的 malloc()、free()等系列函数，可对 free()后相邻的内存空间进行合并，减少内存碎片化。但是标准 C 库无法对堆使用量进行精确统计，只能记录其使用最大值。

获取标准 C 库中堆信息（堆起始地址、堆结束地址、堆分配的最大地址）的 API 如下：

```
void heap_get_space(uint8_t **start, uint8_t **end, uint8_t **current);
```

SDK 提供了获取以上信息的命令，示例如下：

```
$ heap space
<ACK> 200 sram heap total 220680, max use 44064, min free 176616, [0x215df8, 0x220a18, 0x24bc00)
```

如果使能了 PSRAM，则可通过以下命令获取 PSRAM 的堆信息：

```
$ psram space
psram heap space [0x1417400, 0x1800000), total size 4099072 Bytes, free size 3833056 Bytes
<ACK> 200 OK
```

3.2 精确堆统计

如果要对堆使用进行精确统计，需要按照以下说明进行配置操作。

表 3-1 精确堆统计配置说明

配置项	配置说明
使能接口	设置说明： 此项配置用于对堆使用进行精确统计。 设置方式： 对于 XR872/XR808 SDK： （1）在“config.mk”中设置：__CONFIG_MALLOC_TRACE ?= y （2）如果使能了 PSRAM，且需要对 PSRAM 堆的使用进行统计，需要在“config.mk”中设置：__CONFIG_PSRAM_MALLOC_TRACE ?= y 对于 XR806 SDK： （1）通过 make menuconfig，使能“Debug options”->“Trace heap memory usage and error”配置选项 （2）如果使能了 PSRAM，且需要对 PSRAM 堆的使用进行统计，可通过 make menuconfig 使能“Debug options”->“Trace psram heap memory usage and error”

配置项	配置说明
	配置选项

完成以上操作后，调用以下 API 可以打印堆使用的详细信息：

表 3-2 打印堆信息接口函数说明

信息项	说明
原型	void heap_trace_info_show(int verbose);
功能	打印所有堆使用情况的简要信息和详细信息（每个内存块的起始地址和大小），检查每个内存块是否存在内存越界。
参数	verbose 含义解释：是否打印堆使用的详细信息
返回值	无

表 3-3 打印 SRAM 堆使用的详细信息接口函数说明

信息项	说明
原型	void sram_heap_trace_info_show(int verbose);
功能	打印 SRAM 堆使用情况的简要信息和详细信息（每个内存块的起始地址和大小），检查每个内存块是否存在内存越界。
参数	verbose 含义解释：是否打印 SRAM 堆使用的详细信息
返回值	无

表 3-4 打印 PSRAM 堆使用的详细信息接口函数说明

信息项	说明
原型	void psram_heap_trace_info_show(int verbose);
功能	打印 PSRAM 堆使用情况的简要信息和详细信息（每个内存块的起始地址和大小），检查每个内存块是否存在内存越界。
参数	verbose 含义解释：是否打印 PSRAM 堆使用的详细信息
返回值	无

SDK 提供了调用 heap_trace_info_show()打印堆信息的命令，示例如下：

```
heap info 0
<ACK> 200 sram total 363836 (355 KB), use 33745 (32 KB), free 330091 (322 KB), max use 35893 (35 KB), min free 327943 (320 KB)
<ACK> 200 psram total 3869520 (3778 KB), use 2102 (2 KB), free 3867418 (3776 KB), max use 2102 (2 KB),
```

```
min free 3867418 (3776 KB)
<<< total heap info >>>
g_mem_sum              35847 (35 KB)
g_mem_sum_max          37995 (37 KB)
g_mem_entry_cnt      208, max 211
<<< sram heap info >>>
g_sram_sum             33745 (32 KB)
g_sram_sum_max         35893 (35 KB)
g_sram_entry_cnt     162, max 165
<<< psram heap info >>>
g_psram_sum            2102 (2 KB)
g_psram_sum_max        2102 (2 KB)
g_psram_entry_cnt 46, max 46
$
$ heap info 1
<ACK> 200 sram total 363836 (355 KB), use 33745 (32 KB), free 330091 (322 KB), max use 35893 (35 KB), min free 327943 (320 KB)
<ACK> 200 psram total 3869520 (3778 KB), use 2102 (2 KB), free 3867418 (3776 KB), max use 2102 (2 KB), min free 3867418 (3776 KB)
<<< total heap info >>>
g_mem_sum              35847 (35 KB)
g_mem_sum_max          37995 (37 KB)
g_mem_entry_cnt      208, max 211
<<< sram heap info >>>
g_sram_sum             33745 (32 KB)
g_sram_sum_max         35893 (35 KB)
g_sram_entry_cnt     162, max 165
0x20eec8, 192
0x20ef98, 176
0x20f050, 104
0x20f168, 68
0x20f1b8, 72
...
0x218050, 88
0x2180b8, 88
0x218118, 88
0x218310, 28
<<< psram heap info >>>
g_psram_sum            2102 (2 KB)
g_psram_sum_max        2102 (2 KB)
g_psram_entry_cnt 46, max 46
0x140fb40, 28
0x140f4c0, 16
0x140f890, 20
```

```
0x140f910, 480
...
0x140ff90, 20
0x1410080, 16
0x14100e0, 16
0x14101a0, 16
```

精确堆统计的原理是采用静态数组跟踪堆中内存块的分配和释放：

（1）每个数组元素占用 8 个字节，默认的数组大小可记录 1024 个内存块信息（总共占用 8192 字节内存）。

（2）数组的大小由“src/debug/heap_trace.c”中的宏“HEAP_MEM_MAX_CNT”指定。

（3）“HEAP_MEM_MAX_CNT”的值需要根据系统中分配使用的内存块个数来确定。如果“HEAP_MEM_MAX_CNT”过小，则可能发生某些内存块的分配无法跟踪记录，导致统计信息有误。当出现该情况时，会打印如下错误信息：

```
[heap ERR] heap_trace_add_entry():299, heap mem count exceed 1024
```

4 内存泄露

系统可以通过调用 malloc()从堆中分配一块内存块，使用结束后需要调用 free()进行释放。如果忘记调用 free()对堆内存块进行释放，则会出现内存泄漏。如果这种情况频繁发生，则会导致系统可用堆空间越来越少。

通过在一段代码的执行前后分别调用 heap_trace_info_show(1)查看堆内存块的详细信息，再对比前后两次的内存块列表，即可检查出是否存在内存泄漏的情况。

5 内存耗尽

如果在申请动态内存时找不到足够大的内存块，malloc()等内存申请函数将返回 NULL 指针，宣告内存申请失败。此时 SDK 会打印错误信息，示例如下：

```
heap exhausted, incr 1032, 0x267d44 >= 0x267c00
```

6 内存多次释放

如果一块 malloc()分配的堆内存，调用 free()进行了多次释放，将会损坏堆空间。这种损坏导致的异常现象不会立刻表现出来，而会延迟到后面的某一次堆使用时才出现，所以，这种错误的代码逻辑会在系统中埋下不可预知的严重隐患，必须杜绝。

SDK 在开启精确堆统计功能后，可以检测出内存多次释放的问题。例如以下代码运行后，将会出现如下所示的错误信息：

```
void heap_double_free(void)
{
    uint8_t *buf = malloc(16);
    printf("buf %p\n", buf);
    free(buf);
    free(buf);
}
```

```
buf 0x210950
[heap ERR] heap_trace_delete_entry():346, heap mem entry (0x210950) missed
```

7 内存越界

如果向堆内存块中写入超过其长度的数据时，就发生内存写越界的情况，导致堆内存链表错误或者其他内存块被改写。这种损坏导致的异常现象不会立刻表现出来，而会延迟到后面的某一次内存使用时才出现，所以，这种错误的代码逻辑会在系统中埋下不可预知的严重隐患，必须杜绝。

SDK 在开启精确堆统计功能后，可以对堆内存块的写越界进行检查。检查原理如下：

（1）从堆中分配内存的时候，多分配 4 个字节。

（2）在这多分配的 4 个字节（处于内存块的末尾）中填入 4 个魔数：0x4a, 0x5b, 0x6c, 0x7f

（3）调用 free()释放内存块时，检查内存块末尾的 4 字节魔数是否改变；如果改变了，则表示发生了内存写越界，并打印出错信息。

（4）调用 heap_trace_info_show()时，将会检查所有堆内存块末尾的 4 字节魔数是否改变；如果改变了，将会打印出错信息。

例如以下代码运行后，将会出现如下代码中所示的错误信息：

```
void heap_overflow(void)
{
    char *buf = malloc(8);
    strcpy(buf, "12345678");
    heap_trace_info_show(0);
    free(buf);
}
```

```
<<< total heap info >>>
g_mem_sum          37999 (37 KB)
g_mem_sum_max      37999 (37 KB)
g_mem_entry_cnt    211, max 211
<<< sram heap info >>>
g_sram_sum         35897 (35 KB)
g_sram_sum_max     35897 (35 KB)
g_sram_entry_cnt   165, max 165
[heap ERR] heap_mem_show():142, mem (0x210928) corrupt
<<< psram heap info >>>
g_psram_sum        2102 (2 KB)
g_psram_sum_max    2102 (2 KB)
g_psram_entry_cnt 46, max 46
[heap ERR] heap_trace_delete_entry():313, mem f (0x210928, 8) corrupt
```

8 内存跟踪

如果系统中存在内存泄漏、内存越界等情况，通过以上手段可以发现。但是，如果在系统比较复杂的情况下，想要进一步定位哪段代码出现了问题，出现错误的内存块属于哪个模块等，也不是一件容易的事情。SDK 提供了两种内存跟踪手段，可以辅助定位类似问题。

8.1 方式一

该方式可以实时打印出内存申请与内存释放的信息。启动该手段，需要按以下方式进行配置。

表 8-1 开启内存申请与释放打印信息配置说明

配置项	配置说明
内存申请与内存释放打印使能	设置说明： 此项配置用于开启内存申请与内存释放的打印信息。 设置方式： 在“src/debug/heap_trace.c”中设置宏“HEAP_MEM_DBG_ON”为“1”。 设置示例： 定义内存块的跟踪条件。例如要打印大小大于 16 字节的堆内存分配、释放的情况，定义如下： #define HEAP_MEM_DBG_ON 1 #define HEAP_MEM_DBG_MIN_SIZE 16 #define HEAP_MEM_IS_TRACED(size) (size > HEAP_MEM_DBG_MIN_SIZE)

开启内存跟踪功能后，满足跟踪条件的内存块在分配、释放时，都会有相关的打印输出。例如下面代码执行后的输出如下所示：

```
void heap_trace(void)
{
    printf("heap trace begin\n");
    uint8_t *buf = malloc(32);
    printf("buf %p\n", buf);
    free(buf);
    printf("heap trace end\n");
}
```

```
heap trace begin
[heap] m (0x218440, 32)
buf 0x218440
[heap] f (0x218440, 32)
heap trace end
```

结合内存跟踪的打印信息和系统运行过程中的其他打印信息，可粗略定位内存的分配和释放是在哪些上下文代码中执行的，据此再进行进一步分析和定位。

8.2 方式二

利用 backtrace 可以记录下每次内存分配时的函数调用关系。启动该 debug 手段，需要按以下方式进行配置。

表 8-2 启用 backtrace 配置说明

配置项	配置说明
backtrace 使能	设置说明： 此项配置用于启用 backtrace，记录下每次内存分配时的函数调用关系。 设置方式： 对于 XR872/XR808 SDK： 在“config.mk”中设置： CONFIG_BACKTRACE ?= y 对于 XR806 SDK： 通过 make menuconfig 使能“Debug options”->“backtrace”配置选项

使用 backtrace 进行内存跟踪的原理是采用静态数组记录堆中内存块分配时的函数调用堆栈帧数据。与其相关的宏定义有：

```
#define BACKTRACE_COUNT     4
#define BACKTRACE_OFFSET    2
```

其中

（1）BACKTRACE_COUNT 定义记录的函数调用个数

（2）BACKTRACE_OFFSET 是 backtrace 的偏移值

启动 backtrace 后，当调用函数 heap_trace_info_show()、内存多次释放、内存越界时，都会有相关的函数调用堆栈帧数据显示出来。函数 heap_trace_info_show()显示的信息如下所示：

```
$ heap info 1
<ACK> 200 sram total 346596 (338 KB), use 33745 (32 KB), free 312851 (305 KB), max use 35893 (35 KB), min free 310703 (303 KB)
<ACK> 200 psram total 3869520 (3778 KB), use 2102 (2 KB), free 3867418 (3776 KB), max use 2102 (2 KB), min free 3867418 (3776 KB)
<<< total heap info >>>
g_mem_sum               35847 (35 KB)
g_mem_sum_max           37995 (37 KB)
g_mem_entry_cnt     208, max 211
<<< sram heap info >>>
```

```
g_sram_sum              33745 (32 KB)
g_sram_sum_max       35893 (35 KB)
g_sram_entry_cnt    162, max 165
0x213220, 192
backtrace:0x207934 0xbb28 0x201260 0x201202
0x2132f0, 176
backtrace:0x207934 0xbb28 0x201260 0x201202
0x2133a8, 104
backtrace:0x207934 0xbb28 0x201260 0x201202
0x213420, 68
backtrace:0x207934 0xbb56 0x201260 0x201202
...
0x21a768, 88
backtrace:0x207934 0x1a254 0x1a2a2 0x2072b6
0x21a708, 88
backtrace:0x207934 0x1a254 0x1a2a2 0x2072b6
0x21a6d0, 44
backtrace:0x207934 0x1bd24 0x2076a6 0x4b371e
<<< psram heap info >>>
g_psram_sum             2102 (2 KB)
g_psram_sum_max      2102 (2 KB)
g_psram_entry_cnt 46, max 46
0x140f4c0, 16
backtrace:0x204c5e 0x486bac 0x48c870 0x48b318
0x140f4f0, 336
backtrace:0x204c5e 0x486bac 0x48c654 0x48b32c
...
0x1410200, 16
backtrace:0x204c5e 0x436f84 0x4d278c 0x4d28fa
0x1410230, 16
backtrace:0x204c5e 0x436f84 0x4d28fa 0x209d0a
```

内存多次释放显示的信息如下所示：

```
[heap ERR] heap_trace_delete_entry():346, heap mem entry (0x214cd0) missed
========== back trace ==========
backtrace:0x208d9c
backtrace:0x208f90
backtrace:0x2079ca
backtrace:0x201260
backtrace:0x2012fe
backtrace:0x1c01e
backtrace:0xfffffff9
==========       End       ==========
```

内存越界显示的信息示例：

```
<<< total heap info >>>
g_mem_sum          36501 (35 KB)
g_mem_sum_max      38645 (37 KB)
g_mem_entry_cnt    144, max 146
<<< sram heap info >>>
g_sram_sum         32897 (32 KB)
g_sram_sum_max     35041 (34 KB)
g_sram_entry_cnt   132, max 134
[heap ERR] heap_mem_show():142, mem (0x211ac0) corrupt
backtrace:0x205e78 0x201230 0x20124e 0x2013ca
<<< psram heap info >>>
g_psram_sum        3604 (3 KB)
g_psram_sum_max    3604 (3 KB)
g_psram_entry_cnt 12, max 12
```

利用 objdump 反汇编，或 gcc 的工具，可以根据这些函数调用堆栈帧数据回溯得到对应的函数调用列表。比如使用 arm-none-eabi-addr2line 对这些数据进行分析，分析结果示例如下：

```
Exdroid88:~$ ./tools/gcc-arm-none-eabi-4_9-2015q2/bin/arm-none-eabi-addr2line
-pfiCe ./appos/project/test/memory_test/gcc/memory_test.axf 0x205e78
__wrap__malloc_r at /home/appos/src/libc/wrap_malloc.c:89
Exdroid88:~$ ./tools/gcc-arm-none-eabi-4_9-2015q2/bin/arm-none-eabi-addr2line
-pfiCe ./appos/project/test/memory_test/gcc/memory_test.axf 0x201230
heap_overflow at
/home/appos/project/test/memory_test/gcc/../../../../project/test/memory_test/main.c:37
Exdroid88:~$ ./tools/gcc-arm-none-eabi-4_9-2015q2/bin/arm-none-eabi-addr2line
-pfiCe ./appos/project/test/memory_test/gcc/memory_test.axf 0x20124e
main at /home/appos/project/test/memory_test/gcc/../../../../project/test/memory_test/main.c:46
Exdroid88:~$ ./tools/gcc-arm-none-eabi-4_9-2015q2/bin/arm-none-eabi-addr2line
-pfiCe ./appos/project/test/memory_test/gcc/memory_test.axf 0x2013ca
main_task at
/home/appos/project/test/memory_test/gcc/../../../../project/common/startup/gcc/retarget_main.c:39
```

这里清晰明了地指出了出错代码的函数调用关系，错误的代码位于 main.c 的第 37 行，函数 heap_overflow()里，以及 main.c 的 46 行，函数 main 里。

著作权声明

商标声明

免责声明