死机Bug分析方法

Bin.ji

# 死机Bug的分析工具及使用

Lianxiang开发了一组工具可以用来分析bug的callback，stack， memory。

运行trace32cmm\_auto.bat，很多问题就可以直接看到从thread\_entry到死机位置的完整的callback。

运行t32\_stack\_maker.bat，可以生成除当前task以外的其余task的callback，在分析watchdog的时候很有用。

mem\_statist.bat是分析内存的工具，不常用。

dbg\_stack\_make\_call\_list.bat是暴力推栈的工具，不常用。

使用工具以及分析bug的时候要注意log和版本的匹配（有时候因为测试人员搞错了会导致log和版本没有匹配上），推导的时候就会发现一片混乱。检查log和版本是否匹配方法是：

比较ass文件中打印的版本信息时间和axf文件的生成时间，最多相差几分钟，相差时间过大就是不匹配的。例如Bug 2830下面的比较说明是匹配的。

Current Version:

Platform Version: MOCOR\_W12.19\_Debug

sc6530\_128X64\_320X480PDA2

Project Version: SC6530\_PDA

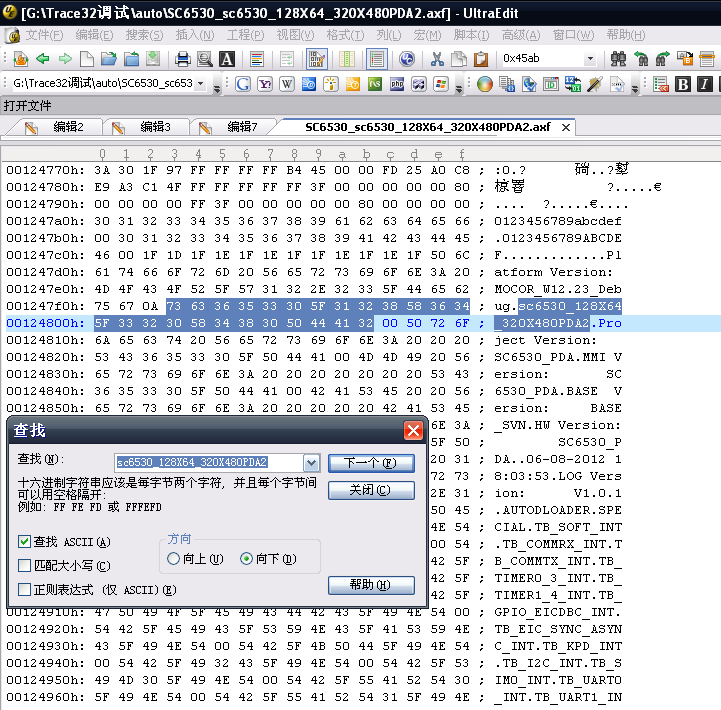
BASE Version: BASE\_SVN

HW Version: SC6530\_PDA

05-04-2012 17:32:05



另外，axf文件的时间也并不一定准确，最准确的可以查看axf二进制文件，在里面查找工程名称（sc6530\_128X64\_320X480PDA2），下面就有生成时间，ass中打印时间就是读取的此时间。



# 死机Bug分析的一般步骤

通常，运行trace32cmm\_auto.bat，很多问题就可以直接看到从thread\_entry到死机位置的完整的callback，死机原因基本上就清楚了。

如果无法看到（很多原因，比如bug异常，ass未打印app task的信息），那么可以采用下面的步骤分析（以bug2285为例）：

察看s\_static\_thread\_table变量thread\_id=21的t\_p\_app内容：

s\_static\_thread\_table =

...

(block\_type = 537006593, thread\_id = 6, thread = (tx\_thread\_id = 1414025796, tx\_run\_count = 12533, tx

(block\_type = 537006593, thread\_id = 9, thread = (tx\_thread\_id = 1414025796, tx\_run\_count = 123, tx\_st

(block\_type = 537006593, thread\_id = 11, thread = (tx\_thread\_id = 1414025796, tx\_run\_count = 2, tx\_sta

(

block\_type = 537006593,

thread\_id = 21,

thread = (

tx\_thread\_id = 1414025796,

tx\_run\_count = 884479,

tx\_stack\_ptr \_=\_0x04032E80,

tx\_stack\_start = 0x04030684,

tx\_stack\_end = 0x0403367F,

tx\_stack\_size = 12284,

tx\_stack\_ptr记录的是当前栈指针（task切换前的值，跟死机时的位置不完全对得上），可以在data.dump中查看tx\_stack\_ptr=0x04032E80。如下所示，

\_\_\_address\_\_\_\_|\_\_\_\_\_\_\_\_0\_\_\_\_\_\_\_\_4\_\_\_\_\_\_\_\_8\_\_\_\_\_\_\_\_C\_0123456789ABCDEF

SD:04032E50| 00000001 000248CF 00000004 045AB848 .....H......H.Z.

SD:04032E60| 00000040 045AB730 00002E53 045AB830 @...0.Z.S...0.Z.

SD:04032E70| 0000000D 00024C6F 00000001 04030000 ....oL..........

SD:04032E80| 00004000 04030000 00004000 00000000 .@.......@......

SD:04032E90| 0000000D 00006E7F 04030000 0000000C .....n..........

SD:04032EA0| 00000000 00006FE9 000001FF 0457B6D0 .....o........W.

SD:04032EB0| 00800000 00000000 00000001 0000747F .............t..

SD:04032EC0| 00000000 40000000 04000000 0000A000 .......@........

SD:04032ED0| 000ECA60 00000033 0457B4DC 00000000 `...3.....W.....

SD:04032EE0| 00000001 00002953 00000033 000012D5 ....S)..3.......

SD:04032EF0| 00000000 00000000 00000000 00002FF0 ............./..

SD:04032F00| 000010EB 04032EF0 00000001 00002F48 ............H/..

并从此地址开始往stack\_end方向找貌似函数地址（0x08\*\*，其他）的值，并且将其作为pc，相邻的地址作为r13代入尝试（注意pc就是r13-4存放的值）。这是反向推栈的办法（tx\_stack\_ptr -> tx\_stack\_end），也可以采用正向推栈的办法（tx\_stack\_end-> tx\_stack\_ptr）。总之，我们找到了一个值：r13=040330B0，pc=08AB2243。将其代入能够获得完整的callback（当然，仍需要pushstack正向和反向验证：正向验证Indirect List看到的就是下一个函数地址，反向验证下一个函数开头寄存器压栈数目反推能够回到上一个函数的r13返回地址）

-000|MMIAPIISTYLESEARCH\_HandleSearchWinMsg(win\_id = 1, ?, ?)

-001|MMIIDLE\_IStyleHandleMsg(win\_id = 1, msg\_id = 57370, param = 0x040331C8)

-002|IdleWin\_HandleMsg(win\_id = 1, msg\_id = 57370, param = 0x040331C8)

-003|MMK\_RunWinProc(?, msg\_id = 57370, param = 0x040331C8)

-004|MMK\_DispatchToHandle(?, ?, ?)

-005|MMK\_SendMsg(?, msg\_id = 57370, param\_ptr = 0x040331C8)

-006|GUICTRL\_SendNotifyEx(?, code = 57370, notify\_ptr = 0x040331C8)

-007|SendNeedItemDataMsg(list\_ptr = 0x04555BFC, item\_index = 15)

-008|DrawAllItems(list\_ptr = 0x04555BFC, lcd\_dev\_info\_ptr = 0x04033258)

-009|DisplayListCtrl(list\_ptr = 0x04555BFC, inner\_state = 31)

-010|GUILIST\_HandleMsg(?, ?, ?)

-011|MMK\_RunCtrlProc(?, msg\_id = 61494, param = 0x0)

-012|ControlTreeNodeHandleEvent(?, ?, ?)

-013|MMK\_DispatchToAllTreeNode(?, func = 0x0805D76B, msg\_id = 61494, param = 0x0

-014|MMK\_ProcSpecialWinMsg(win\_handle = 16908289, ?, param = 0x0)

-015|MMK\_DispatchToHandle(handle = 16908289, msg\_id = 61477, param\_ptr = 0x0)

-016|MMK\_SendMsg(?, msg\_id = 61477, param\_ptr = 0x0)

-017|HandleFormNotifyResize(resize\_rect = (left = 0, top = 0, right = 0, bottom

-018|FormHandleMsg(?, ?, param = 0x04033450)

-019|MMK\_RunCtrlProc(?, msg\_id = 1, param = 0x04033450)

-020|MMK\_DispatchToHandle(handle = 266731711, msg\_id = 1, param\_ptr = 0x04033450

-021|MMK\_SendMsg(?, msg\_id = 1, param\_ptr = 0x04033450)

-022|GUICTRL\_SendNotifyEx(?, code = 1, notify\_ptr = 0x04033450)

-023|DestroyImCtrl(?, ?)

-024|HandleEditLoseActive(?, edit\_ctrl\_ptr = 0x04564544)

-025|EditHandleMsg(?, msg\_id = 4, param = 0x04033520)

-026|MMK\_RunCtrlProc(?, msg\_id = 4, param = 0x04033520)

-027|MMK\_DispatchToHandle(handle = 266862642, msg\_id = 4, param\_ptr = 0x04033520

-028|MMK\_SendMsg(?, msg\_id = 4, param\_ptr = 0x04033520)

-029|GUICTRL\_SendNotifyEx(?, code = 4, notify\_ptr = 0x04033520)

-030|GUICTRL\_SendNotify(ctrl\_handle = 278724673, code = 4)

-031|ImHandleMsg(?, ?, ?)

-032|MMK\_RunCtrlProc(?, msg\_id = 61491, param = 0x0)

-033|MMK\_SetActiveCtrl(?, is\_need\_paint = 0)

-034|MMK\_SetAtvCtrlEx(?, ?, is\_need\_paint = 0)

-035|DispatchMSGTpDown(?, ?)

-036|MMK\_DispatchMSGTp(?)

-037|MMK\_DispatchExtSig(?, ?)

-038|thread\_entry\_P\_APP(?, ?)

-039|ThreadEntry(?)

-040|tx\_thread\_shell\_entry()

---|end of frame

到了这一步，我们已经获得了大量信息：完整的callback告诉我们大致的死机步骤和场景，参数（但并不一定准确）。接下去可以比较当前r13和tx\_stack\_ptr的差距，将callback向更深推进，如果无法推进就进行参数代入试验（后文有介绍）。结合s\_zorder\_system和log信息我们还可以做大量的分析和模拟器试验（后文有介绍）。

# 死机Bug的初步分析

按照assert的不同提示信息，一般可以分为以下几类。

## File:sci\_mem.c Line:536 ASSERT(ASSERT:Error 0xff, Normal error, mmiim\_touch\_ime\_cstar.c line=5322, param=0x1B30)

（例如bug1102）

类似于这样的问题是最普通的Assert，直接看打印的信息就可以知道出错的文件和代码行。

## File:boot\_mode.c Line: 144 PASSERT(0) > Unexpectly PC set to 0

Unexpectly PC set to 0对应的pc跳转到0，一般情况发生在函数指针跳转到0的时候，这类问题通过callback可以很容易分析出来原因。但是也有很特殊的例子，例如bug3341，后面会提到。

## Exception at 0x00008008 ASSERT(Reserved exception handler !)

（例如bug2285）

这种问题说明最后pc跳转到0x00008008这个异常地址。这种情况如果已经找到了临近死机的汇编代码，可以找找那里有没有跳转到寄存器的指令（例如bl r3），对应于C语言一定有一个函数指针的执行。

## Exception at 0x000f2c4c

ASSERT(Abort exception handler !)

Abort fault(DFSR:0x00000005): Translation fault, Domain invalid !

Fault address :0xb194423c

（例如bug3009）

虽然是很小的地址，但是跟上一类问题不同，这确实是正常的函数地址，只不过往往是系统库函数，运行trace32cmm\_auto.bat可以直接看到callback。

## Exception at 0x08655b4a

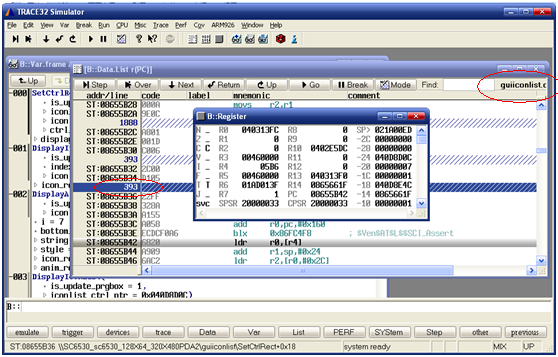
ASSERT(Abort exception handler !)

Abort fault(DFSR:0x00000001): Alignment fault, Domain invalid !Fault address :0x000005b6

（例如bug1557，bug1806）

0x08开头的地址一般是正常的函数地址，只要能找到对应的异常位置，原因基本上很明朗。

例如bug1557，pc=0x08655b4a-8=0x08655b42就是出错位置，r4=0x05b6，从该地址读数据异常（地址没有4字节对齐），显然0x05b6是个非法地址（但是如果他碰巧4字节对齐可能就过去了）。同时我们也可以看到对应的C代码文件名和行号。



## Exception at 0x09045f4e

ASSERT(Abort exception handler !)

Abort fault(DFSR:0x0000000d): Permission fault, Domain valid !

Fault address :0x0000006c

（例如bug2573）

0x09开头的地址也是跟0x08开头的地址一样，一般是正常的函数地址，分析方法一样。

## Exception at 0x044d11a8

ASSERT(Abort exception handler !)

Abort fault(DFSR:0x0000000d): Permission fault, Domain valid !

Fault address :0x00007824

（例如bug2708，bug1916，bug1571）

0x04开头的地址往往是跑飞了（正常的函数地址很少0x04开头，少数如WRE动态加载code，还有sci\_mem.c的函数地址也有0x04开头例如bug3008），跑飞exception原因很多，很多是内存覆盖引起的。

## Div 0, retarget

（例如bug1817）

发生除0错误，分析出callback以及最终死机位置就行了。

## [Monkey test]自动测试 黑屏， 不响应，手动assert

（例如bug1405）

类似于这样的问题属于黑屏（定屏）无响应问题，需要一个单独章节讲述。

## sci\_mem.c Line: 177 [0x040ddd54] had been freed pointer!

（例如bug2006）

使用已释放指针问题，需要查一下流程。少数是因为内存覆盖或内存管理出错。

## ASSERT(ASSERT: Error 0x10,No memory, unable to allocate,mmikm\_playlist.c, line=2393, param=0x1B0D0)

（例如bug1035）

内存不足。这类问题首先看一下是否存在内存泄露（某块内存重复打印多次），如果不存在内存泄露，就要看一下System Space被谁占用着以及当前场景是什么（s\_zorder\_system），能否节省内存。

## File: watchdog.c Line: 312 PASSERT(SCI\_FALSE)Task APP timeout

（例如bug1321）

这类问题只要用lianxiang的工具把t\_p\_app的callback导出来分析一下就明白了（一种可能是t\_p\_app在get\_mutex或者get semphore的时候挂住了，另一种可能是t\_p\_app的callback中有一处死循环未退出）。

## File:RTOS/source/src/c/threadx\_os.cLine:677ASSERT(ASSERT: Error 0xb,The queue was full)

（例如bug2930）

Queue full问题分析方法跟watchdog.c有些类似。用lianxiang的工具把t\_p\_app的callback导出来看一下。

## File: prod\_param.c Line: 731PASSERT(0)Stack Overflow,thread:0x45c0994,sp overflow addr:0x402a928,thread ID:0x15,Tx Name:T\_P\_APP

（例如NEWMS00199242）

导致T\_P\_APP stack overflow的原因一般有3种：Callback函数调用次数过深（或者出现循环），局部变量Size过大，有些代码执行上发生异常（指向栈变量的指针++，--越界）。

# 内存覆盖问题分析（展开）

bug2708：

Exception at 0x044d11a8

ASSERT(Abort exception handler !)

Abort fault(DFSR:0x0000000d): Permission fault, Domain valid !

Fault address :0x00007824

可以在ass文件中找到2处表明内存破坏的迹象，0x449463c处内存被破坏。具体原因需要进一步分析。

>

17421409 316 mmisrvaud.c (Line 2503),addr:45a53b4

>

17421410 12 mmi\_link.c (Line 281),addr:45b1c9c

>

0 0 馃?馃?馃?馃?馃?馃?馃?馃宕? (Line 0),addr:44952d4

>

memory is corrupted, abnormal termination

...

================== System Space Information ==============

>

Begin Address End Address Total\_Num Avail\_Num Max\_Used Threshold

>

>

0x449463c 0x0 -71910972 ALLOC 馃?馃?馃?馃?馃?馃?馃?馃宕?(0)

>

0x0 0xe59ff018 -442503144 ALLOC (-442503144)

>

Heap Error Start:0x0 , file name:, Line:-442503144

>

bug1916：

Exception at 0x0400596a

ASSERT(Abort exception handler !)

Abort fault(DFSR:0x00000005): Translation fault, Domain invalid !

Fault address :0x10821082

执行SCI\_Free(s->p\_work\_mem);时发现s->p\_work\_mem所在的地址被覆盖

LOCAL void \_ISP\_ServiceDeInit(void)

{

ISP\_SERVICE\_T \*s = &s\_isp\_service;

s->is\_slice\_alone = 0;

if(s->mem\_ctrl)

{

SCI\_GetMutex(s->mem\_ctrl, SCI\_WAIT\_FOREVER);

}

if(s->mem\_alloced\_by\_myself)

{

SCI\_Free(s->p\_work\_mem);

s->p\_work\_mem = SCI\_NULL;

s->mem\_alloced\_by\_myself = SCI\_FALSE;

s->work\_mem\_len = 0;

}

-000|SCI\_DumpAllocatedMemInfo()

| str = 0x0459F3AC

| name = (0x6D, 0x6D, 0x69, 0x5F, 0x6C, 0x69, 0x6E, 0x6B, 0x2E, 0x63, 0x0,

| file\_ptr = 0x0

| new\_file\_ptr = 0x0

| time\_out = 0x0

| search\_cnt = 0x0

| tmpptr = 0x0

-001|TXAS\_Cmd\_DumpAllocatedMemInfo(

| ?,

| ?,

| ?)

-002|TXAS\_SystemAssert(

| exp = 0x045CF8B0,

| file = 0x04003E04,

| line = 0xF8,

| ?)

| assert\_mode = 0x1

| cur\_sp = 0x045CF870

| cur\_lr = 0x10EB

| cur\_pc = 0x2FF0

-003|\_\_CheckMemDebugInfo(

| ?,

| file = 0x0,

| line = 0x0)

| header\_ptr = 0x042C7F24

| size = 0x00025818

-004|SCI\_Free(

| ?)

-005|ISP\_ServiceDeInit()

| s = 0x045CE728 -> (

| module\_addr = 0x20200000,

| service = 0x7,

| state = 0xE0,

| cap\_input\_image\_format = 0x0,

| cap\_input\_image\_patten = 0x0,

| cap\_sensor\_if\_mode = 0x0,

| cap\_sensor\_if\_endian = 0x0,

| vsync\_polarity = 0x1,

| hsync\_polarity = 0x1,

| thre\_start = 0x0,

| thre\_end = 0x0,

| fix\_mode\_eb = 0x0,

| b\_is\_atv = 0x0,

1. 内存不足问题分析（展开）。

（例如bug1035）

当前剩余可用内存=233544=230KB，已经不太够了。sci\_mem.c分配了33次49K左右的内存，可能存在内存泄漏。

================== System Space Information ==============

>

Begin Address End Address Total\_Num Avail\_Num Max\_Used Threshold

>

0x40ce17c 0x451a17c 4505600 233544 4469412 4505600

>

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

>

Start\_Addr End\_Addr Size State File\_Name(Line)

>

0x40ce17c 0x40d3314 0x5198 ALLOC threadx\_os.c(1184)

>

0x40d3314 0x40d84ac 0x5198 ALLOC threadx\_os.c(1184)

>

0x40d84ac 0x41235d0 0x4b124 ALLOC mem\_prod.c(171)

>

0x41235d0 0x4127df4 0x4824 ALLOC os\_pmalloc.c(84)

>

0x4127df4 0x4133e18 0xc024 ALLOC sci\_mem.c(349)

>

0x4133e18 0x413fe3c 0xc024 ALLOC sci\_mem.c(349)

>

0x413fe3c 0x4141184 0x1348 FREE 馃?馃?馃?馃?馃?馃?馃?馃?j(0)

>

0x4141184 0x414d1a8 0xc024 ALLOC sci\_mem.c(349)

>

0x414d1a8 0x415241c 0x5274 FREE fat\_folder.c(40)

>

0x415241c 0x415e440 0xc024 ALLOC sci\_mem.c(349)

>

0x415e440 0x416a464 0xc024 ALLOC sci\_mem.c(349)

>

0x416a464 0x4173800 0x939c ALLOC ui\_layer.c(2571)

>

0x4173800 0x417611c 0x291c FREE 馃?馃?馃?馃?馃?馃?馃?馃?j(0)

>

0x417611c 0x4182140 0xc024 ALLOC sci\_mem.c(349)

>

0x4182140 0x418e164 0xc024 ALLOC sci\_mem.c(349)

>

0x418e164 0x419a188 0xc024 ALLOC sci\_mem.c(349)

>

0x419a188 0x41a61ac 0xc024 ALLOC sci\_mem.c(349)

>

0x41a61ac 0x41b21d0 0xc024 ALLOC sci\_mem.c(349)

>

0x41b21d0 0x41bb56c 0x939c ALLOC ui\_layer.c(2571)

>

0x41bb56c 0x41be118 0x2bac FREE 馃?馃?馃?馃?馃?馃?馃?馃?j(0)

>

0x41be118 0x4209240 0x4b128 ALLOC ui\_layer.c(2571)

>

0x4209240 0x420d4ac 0x426c FREE fat\_folder.c(40)

>

0x420d4ac 0x42194d0 0xc024 ALLOC sci\_mem.c(349)

>

0x42194d0 0x42254f4 0xc024 ALLOC sci\_mem.c(349)

>

0x42254f4 0x422ba1c 0x6528 ALLOC ui\_layer.c(2571)

>

0x422ba1c 0x422c4f8 0xadc FREE ui\_layer.c(-1521179276)

>

0x422c4f8 0x423086c 0x4374 ALLOC mmiwidget\_sns.c(1155)

>

0x423086c 0x423c890 0xc024 ALLOC sci\_mem.c(349)

>

0x423c890 0x4249884 0xcff4 ALLOC mmivirtualarray.c(380)

>

0x4249884 0x424c9f8 0x3174 FREE 馃?馃?馃?馃?馃?馃?馃?馃?j(0)

>

0x424c9f8 0x42599ec 0xcff4 ALLOC mmivirtualarray.c(380)

>

0x42599ec 0x425bb94 0x21a8 FREE mmivirtualarray.c(1325400064)

>

0x425bb94 0x425fbb8 0x4024 ALLOC vf\_malloc\_stdlib.c(191)

>

0x425fbb8 0x4260db4 0x11fc FREE 馃?馃?馃?馃?馃?馃?馃?馃?j(0)

>

0x4260db4 0x4272ee8 0x12134 ALLOC mmipb\_datalist.c(776)

>

0x4272ee8 0x427ef0c 0xc024 ALLOC sci\_mem.c(349)

>

0x427ef0c 0x4290b14 0x11c08 ALLOC ui\_layer.c(2571)

>

0x4290b14 0x42a9be4 0x190d0 ALLOC ui\_layer.c(2571)

>

0x42a9be4 0x42ae52c 0x4948 FREE 馃?馃?馃?馃?馃?馃?馃?馃?j(0)

>

0x42ae52c 0x42bb520 0xcff4 ALLOC mmivirtualarray.c(380)

>

0x42bb520 0x42d3748 0x18228 ALLOC ui\_layer.c(2571)

>

0x42d3748 0x42dcae4 0x939c ALLOC ui\_layer.c(2571)

>

0x42dcae4 0x42e267c 0x5b98 FREE 馃?馃?馃?馃?馃?馃?馃?馃?j(0)

>

0x42e267c 0x42ee6a0 0xc024 ALLOC sci\_mem.c(349)

>

0x42ee6a0 0x42fa6c4 0xc024 ALLOC sci\_mem.c(349)

>

0x42fa6c4 0x43066e8 0xc024 ALLOC sci\_mem.c(349)

>

0x43066e8 0x4306824 0x13c FREE sci\_mem.c(-1543503872)

>

0x4306824 0x4312848 0xc024 ALLOC sci\_mem.c(349)

>

0x4312848 0x431e86c 0xc024 ALLOC sci\_mem.c(349)

>

0x431e86c 0x432a890 0xc024 ALLOC sci\_mem.c(349)

>

0x432a890 0x43371b8 0xc928 ALLOC ui\_layer.c(2571)

>

0x43371b8 0x433bd20 0x4b68 ALLOC ui\_layer.c(2571)

>

0x433bd20 0x433c2e0 0x5c0 FREE 馃?馃?馃?馃?馃?馃?馃?馃?j(0)

>

0x433c2e0 0x4348304 0xc024 ALLOC sci\_mem.c(349)

>

0x4348304 0x4354328 0xc024 ALLOC sci\_mem.c(349)

>

0x4354328 0x4354c64 0x93c FREE 馃?馃?馃?馃?馃?馃?馃?馃?j(0)

>

0x4354c64 0x4360c88 0xc024 ALLOC sci\_mem.c(349)

>

0x4360c88 0x436ccac 0xc024 ALLOC sci\_mem.c(349)

>

0x436ccac 0x43b2ed0 0x46224 ALLOC ui\_layer.c(2571)

>

0x43b2ed0 0x43b89ac 0x5adc FREE 馃?馃?馃?馃?馃?馃?馃?馃?j(0)

>

0x43b89ac 0x43c49d0 0xc024 ALLOC sci\_mem.c(349)

>

0x43c49d0 0x43c691c 0x1f4c FREE sci\_mem.c(-268475392)

>

0x43c691c 0x43d2940 0xc024 ALLOC sci\_mem.c(349)

>

0x43d2940 0x43dac58 0x8318 FREE s3d\_rotation\_se.c(3418)

>

0x43dac58 0x43e6c7c 0xc024 ALLOC sci\_mem.c(349)

>

0x43e6c7c 0x43f2ca0 0xc024 ALLOC sci\_mem.c(349)

>

0x43f2ca0 0x4488dc8 0x96128 ALLOC ui\_layer.c(2571)

>

0x4488dc8 0x44c76f0 0x3e928 ALLOC ui\_layer.c(2571)

>

0x44c76f0 0x44caf1c 0x382c FREE 馃?馃?馃?馃?馃?馃?馃?馃?j(0)

>

0x44caf1c 0x44d6f40 0xc024 ALLOC sci\_mem.c(349)

>

0x44d6f40 0x44e2f64 0xc024 ALLOC sci\_mem.c(349)

>

0x44e2f64 0x44eef88 0xc024 ALLOC sci\_mem.c(349)

>

0x44eef88 0x44fafac 0xc024 ALLOC sci\_mem.c(349)

>

0x44fafac 0x4504348 0x939c ALLOC ui\_layer.c(2571)

>

0x4504348 0x4506494 0x214c FREE 馃?馃?馃?馃?馃?馃?馃?馃?j(0)

>

0x4506494 0x45124b8 0xc024 ALLOC sci\_mem.c(349)

>

0x45124b8 0x45189e0 0x6528 ALLOC ui\_layer.c(2571)

>

0x45189e0 0x451a174 0x1794 FREE 琌\*怉?袄z蛗?ip€?鰘糞水?漏((殢.?16300064)

>

# App task Watchdog问题（展开）

（例如bug1321是get\_mutex时挂死的情况）

MMISRVAUD\_GetMutext()挂死了。

-000|tx\_thread\_suspend(?)

-001|tx\_mutex\_get(?, ?)

-002|txe\_mutex\_get(?, ?)

-003|SCI\_GetMutex(?, ?)

-004|MMISRVAUD\_GetMutext(mutex\_ptr = 0x0454F974)

-005|SrvCmdFunc(param = 0x0403169C, res\_data = 0x04031698)

-006|GetEntityPtr(?)

-007|MMISRVMGR\_IoCtrl(?, data = 67311312, res = 0x0)

-008|MMISRVAUD\_Stop(audio\_handle = 23658496)

-009|ReleaseDCModule()

-010|DCApplet\_HandleEvent(?, ?, ?)

-011|MMK\_RunAppletProc(?, msg\_id = 61458, param = 0x0)

-012|MMK\_DispatchToHandle(handle = 260440155, msg\_id = 61458, param\_ptr = 0x0)

-013|MMK\_SendMsg(?, msg\_id = 61458, param\_ptr = 0x0)

-014|AppletTreeNodeDestruct(?)

-015|MMK\_RemoveTreeNode(?, func = 0x0830864D, ?)

-016|ReturnFirstApplet()

-017|MMK\_ReturnIdleWin()

-018|MMK\_HandleRedKeyProcess()

-019|MMK\_HandlePublicKey(msg\_id = 64280, ?)

-020|HandleMSGKbd(?, ?)

-021|MMK\_DispatchMSGKbd(?)

-022|MMK\_DispatchExtSig(?, ?)

-023|thread\_entry\_P\_APP(?, ?)

-024|ThreadEntry(?)

-025|tx\_thread\_shell\_entry()

---|end of frame

（bug721是死循环未退出的情况）

AddToGroup(header\_group\_ptr = 0x0, ?)

MMK\_TravelActiveCtrl(?, travel\_active = MMK\_TRAVEL\_ACTIVE\_UP)

MMK\_DispatchToHandle(handle = 44892270, ?, ?)

MMK\_DispMsgToFocusWin(msg\_id = 64001, param\_ptr = 0x04021DD4)

MMK\_DispMsgToWin(?, ?)

HandleMSGKbd(keys\_status = 64000, key\_code = 1)

MMK\_DispatchMSGKbd(?)

MMK\_DispatchExtSig(?, ?)

thread\_entry\_P\_APP(?, ?)

ThreadEntry(?)

tx\_thread\_shell\_entry()

AddToGroup中存在一个while循环

LOCAL void AddToGroup(void)

{

...

// group链表里的值肯定是从小到大排序的

while (header\_group\_ptr != (group\_node\_ptr = (MMK\_GROUP\_LIST\_T\*)group\_node\_ptr->group\_node.next))

{

if (group\_node\_ptr->top > ctrl\_rect.bottom) // 由于是排序的，所以当底都比group链表top都小，说明后面肯定不会有重叠的

{

break;

}

else if (group\_node\_ptr->bottom < ctrl\_rect.top) // 在所在group链表的下方，再去找下一个

{

continue;

}

else // 位置重叠了

{

is\_in\_pos\_list = TRUE;

break;

}

}

...

}

在通过查看s\_zorder\_system知道死在哪个窗口，并且询问相关模块负责人了解怎么到达这个窗口，然后在模拟器和手机上重现（发现此问题的必现路径）。通过trace察看死机前操作步骤和task switch分析打点信息也是用来分析watchdog的常用方法。

\*.lst文件过大怎么办？。

用GSplit工具可以将\*.lst文件分割，然后分析最后那块就行了（分析Log一般只要关心最后那段时间跑的trace）。

# Queue full问题（展开）

（例如bug2930）

用工具将app task的callback展开如下，说明此时task挂住了，导致消息过来得不到处理queurfull。还有一类queuefull是因为app task处理消息的速度太慢（例如连续的tp\_move）导致queuefull，解决这类问题一般要优化速度。

-000|tx\_thread\_suspend(

| ?)

-001|tx\_thread\_sleep(

| timer\_ticks = 200)

-002|SCI\_Sleep(

| time\_in\_ms = 200)

-003|GC2015\_set\_anti\_flicker(

| mode = 0)

-004|Sensor\_Ioctl(

| ?,

| arg = 0)

| func\_ptr = 0x000BF8E7

| temp = 784615

| ret\_value = 0

-005|DCAMERA\_PreviewSensorSet()

| preview\_ctl\_info\_ptr = 0x0466FF98

| meter\_set = (0, 0, 0, 73858828, 73858676, 67323844, 67323712, 0)

-006|DCAMERA\_StartPreviewEX(

| ?)

| dc\_info\_ptr = 0x0466FE74

| p\_vt\_enc = 0x0466FF0C

| src\_coor = (rect = (x = 0, y = 0, w = 320, h = 480), rot\_degree = DCAMERA\_ROTATION\_0

| dst\_coor = (rect = (x = 0, y = 0, w = 320, h = 480), rot\_degree = DCAMERA\_ROTATION\_0

| isp\_img\_rect = (x = 0, y = 120, w = 320, h = 240)

| logic\_disp\_rect = (x = 0, y = 120, w = 320, h = 240)

| logic\_lcdc\_rect = (x = 0, y = 0, w = 320, h = 480)

| mem\_size = 0

-007|DCAMERA\_StartPreviewEX(

| param\_ex\_ptr = 0x040347C4)

| actual\_event = 67323808

-008|StartDCPreview()

| ret = DCAMERA\_OP\_SUCCESS

| preview\_param = (lcd\_id = LCD\_ID\_0, lcd\_block\_id = 0, disp\_rect = (x = 0, y = 0, w =

-009|MMIDC\_FlowStart()

| is\_first\_preview = 0

-010|MMIDC\_HandleGetFocus()

-011|HandleCameraWinMsg(

| ?,

| msg\_id = 64286,

| ?)

-012|MMK\_RunWinProc(

| ?,

| msg\_id = 64286,

| param = 0x0403499C)

| win\_id\_name = 0x045FC718

-013|MMK\_DispatchToHandle(

| handle = 277020752,

| msg\_id = 64286,

| param\_ptr = 0x0403499C)

| openwin\_handle\_result = 0

| type = MMI\_HANDLE\_WINDOW

| old\_handle = 16711680

-014|MMK\_DispMsgToFocusWin(

| msg\_id = 64286,

| param\_ptr = 0x0403499C)

-015|MMK\_DispMsgToWin(

| msg\_id = 64286,

| ?)

| result = 0

-016|HandleMSGKbd(

| keys\_status = 64256,

| key\_code = 30)

| multi\_key\_tp\_param = (is\_slide = 0, pre\_tp\_point = (x = 0, y = 0), cur\_tp\_point = (x

-017|MMK\_DispatchMSGKbd(

| ?)

| key\_code = 30

| is\_long\_press = 0

-018|MMK\_DispatchExtSig(

| ?,

| ?)

-019|thread\_entry\_P\_APP(

| ?,

| ?)

| receiveSignal = 0x045A320C

| mmi\_msg = 0x04582E8C

| ticks1 = 0

| ticks2 = 0

| is\_log\_on = 0

| watchdog\_ptr = 0x0458F394

-020|ThreadEntry(

| ?)

| thread\_entry = (entry = 0x08905AD7, argc = 0, argv = 0x0)

-021|tx\_thread\_shell\_entry()

---|end of frame

# 使用已释放指针问题（展开）

（例如bug2794）

有些问题经过分析可以确定为使用已释放指针问题。例如bug2794。

正常的处于alloc状态的内存，mem\_header是完整的。比如随片看一个内存

s\_zorder\_system = (

top\_most\_ptr = 0x0455011C -> (

win\_node\_ptr = 0x045500FC -> (

base\_node = (handle\_type = MMI\_HANDLE\_WINDOW, handle = 24576062),

parent\_applet\_handle = 16711680,

win\_ptr\_=\_0x0456E494,

win\_id = 10,

focus\_child\_win\_ptr = 0x0,

查看win\_ptr\_=\_0x0456E494指向的内存，往前数6个uint32，分别对应下面，MEM\_HEADER\_T的每个成员都可以看到合理的现象（…）。

0456445C ...L......P.\DV.

SD:0456E480| 045500E4 089CD950 0000060D 0000006C ..U.P.......l...

SD:0456E490| 00009D7A

\_\_\_address\_\_\_\_|\_\_\_\_\_\_\_\_0\_\_\_\_\_\_\_\_4\_\_\_\_\_\_\_\_8\_\_\_\_\_\_\_\_C\_0123456789ABCDEF

SD:0456E430| 00000B68 00000030 00009CA0 4154494D h...0.......MITA

SD:0456E440| 08724E4C 00000000 00000000 087249D1 LNr..........Ir.

SD:0456E450| 0462CAA8 00000000 00000000 00000000 ..b.............

SD:0456E460| 00000000 0454E06C 0456408C B1F8FCAA ....l.T..@V.....

SD:0456E470| 4C159610 0000009B 0450D708 0456445C ...L......P.\DV.

SD:0456E480| 045500E4 089CD950 0000060D 0000006C ..U.P.......l...

SD:0456E490| 00009D7A>08A7270D 00000000 00000000 z....'..........

SD:0456E4A0| 00000000 00000001 0177003E 00000000 ........>.w.....

SD:0456E4B0| 00000000 00000000 00000000 00000000 ................

SD:0456E4C0| 40000000 00000204 00000000 00000000 ...@............

SD:0456E4D0| 01DF013F 00000000 01DF013F 00000000 ?.......?.......

SD:0456E4E0| 01DF013F 00000000 013F01DF 00000000 ?.........?.....

而被释放的内存head就可能设置free的flag（刚释放），或者被改成混乱的值（被新申请的内存使用）。

ptTrackBufTimer\_=\_0x045642D4, 使用了已经释放的指针，

\_\_\_address\_\_\_\_|\_\_\_\_\_\_\_\_0\_\_\_\_\_\_\_\_4\_\_\_\_\_\_\_\_8\_\_\_\_\_\_\_\_C\_0123456789ABCDEF

SD:04564290| 00000000 00000001 FFFF0000 0002001A ................

SD:045642A0| 0000000A 0000000A 00010000 00000001 ................

SD:045642B0| FFFF0000 0002001E 00000014 00000005 ................

SD:045642C0| 00020000 00000001 FFFF0000 00020019 ................

SD:045642D0| 00000019>0000000A 00030000 00000001 ................

SD:045642E0| FFFF0000 00020019 00000023 0000000A ........#.......

SD:045642F0| 00040000 00000001 FFFF0000 00000000 ................

SD:04564300| 00000000 00000000 00000000 00000000 ................

-000|TXAS\_SystemAssert(

| exp = 0x0402CB04,

| file = 0x040064BC,

| line = 3214,

| assert\_info\_ptr = 0x0)

| assert\_mode = 1

| cur\_sp = 67291832

| cur\_lr = 4613

| cur\_pc = 13016

| i = 0

-001|SCI\_Assert(

| ?,

| ?,

| ?)

-002|SCI\_DeleteTimer(

| ?)

| status = 21

| \_timer\_ptr = 0x045642D4

-003|thread\_entry\_AUDIO(

| ?,

| ?)

| sig\_in\_ptr = 0x045511A4

| ptAudio = 0x04554B9C -> (

| uiHandleID = 6959521,

| hAudioCodec = 2036385,

| hDevice = 2950305,

| tThreadId = 72800892,

| tAssiThreadId = 72803356,

| tSender = 21,

| ppvSigIn = 0x0456D380,

| ppvSigOut = 0x0,

| unAudioStream = (tAudioFileStream = (fFileHandle = 67318400,

uiFileLength = 32, disk\_denot

| uiTrackNum = 4294967295,

| puiTrackBuffer = 0x0467AD24,

| uiTrackBufferSize = 12000,

| uiUsedTrackBufferSize = 4000,

| uiOutputSamplerate = 8000,

| uiCodecOutputSamplerate = 8000,

| ePlayingFlag = AUDIO\_PLAYING\_FLAG\_STARTSTOP,

| uiCurSrcOffset = 0,

| tCodecProcessResouce = (left\_exp\_data\_ptr = (0x0455FC9C, 0x04561CC4),

right\_exp\_data\_ptr =

| tAudioFormatInfo = (hCodecHandle = 2036385, uiTotalDataLength = 0,

uiTotalTime = 0, uiSamp

| tPlayingInfo = (uiTotalTime = 0, uiCurrentTime = 0, uiTotalDataLength

= 4294967295, uiCurr

| tCodecExtCfgInfo = (uiHighPriority = 74, uiLowPriority = 78,

uiBufFullWatermark = 80, uiBu

| eTrackBufStatus = AUDIO\_TRACKBUF\_EMPTY,

| ptTrackBufTimer\_=\_0x045642D4,

| ptTrackOnTimer = 0x0456432C,

| uiTrackOnTimeLength = 900,

| tStoporpauseSynCallerThreadId = 68300468,

| eStoporpauseSynInterfaceResult = AUDIO\_NO\_ERROR,

| bCurrentStoporpauseOpeSynFlag = 1,

| bDevicePaused = 0,

| bTrackBufStatusNotifyHasClosed = 0,

| bHasEnableDumpPcmFunction = 0,

| pGetSrcCallbackFunc = 0x085D2D2B,

| pNotifyCallbackFunc = 0x0,

| bIsSpecCaculatorOn = 0,

| bEnableDebugDataOutput = 0,

| ucDebugFileHandleCount = 0,

| ucDebugFileLeftRightChannel = 0,

| pbEnableDebugDataOutput = 0x0,

| phDebugFileHandleList = 0x0,

| playing\_info\_mutex = 0x04555034,

| expIfWorkingInfoMutex = 0x0455508C,

| expIfWorkingInfo = 0x04564214,

| writeTrackBufferMutex = 0x045550E4,

| bJustFinishInitExp = 1,

| hClonedHandle = 0,

| res0 = 0,

| res1 = 0)

| result = AUDIO\_NO\_ERROR

| temp\_sig\_in\_ptr = 0x045511A4

-004|ThreadEntry(

| ?)

| thread\_entry = (entry = 0x0004FCCF, argc = 0, argv = 0x0)

-005|tx\_thread\_shell\_entry()

---|end of frame

|

使用已释放指针常见的原因：一是由于同一task内部流程控制的问题，比如全局变量a在某处已经释放，在后面又使用。二是多task通过不可重入函数访问同一全局变量，bug4540就是这样的一个例子。

T\_DCAMERA task和t\_p\_app task同时操作了控件，导致t\_p\_app task释放了控件以后T\_DCAMERA task还在使用控件指针button\_ctrl\_ptr = 0x045AF7CC，于是出错。

由于目前mmk和gui的接口并不完全支持多task重入，请SetReviewOSDCallback注意不要做太多事情，尤其不要同步调用mmk/gui的接口（例如MMK\_SendMsg或其他控件窗口的接口函数）

0x45af7a8 0x45af9c8 544 FREE

T\_DCAMERA task的callback如下：

-000|TXAS\_SystemAssert(

| exp = 0x08CEDC30,

| file = 0x08CEDB8C,

| line = 1025,

| assert\_info\_ptr = 0x0)

| assert\_mode = 1

| cur\_sp = 73030416

| cur\_lr = 4965

| cur\_pc = 13384

| i = 0

-001|SCI\_Assert(

| ?,

| ?,

| ?)

-002|GetParentHandle(

| ctrl\_handle = 0x37BA0079)

-003|GUICTRL\_SendNotifyEx(

| ?,

| code = 2,

| notify\_ptr = 0x045A5BDA)

| result = 0

-004|GuiComDisplayBg(

| ?,

| ?,

| ?,

| ?)

| cross\_rect = (left = 0, top = 0, right = 0, bottom = 0)

-005|DisplayButtonBg(

| button\_ctrl\_ptr = 0x045AF7CC)

| img\_width = 0

| img\_height = 0

| button\_bg = (bg\_type = GUI\_BG\_NONE, shape = GUI\_SHAPE\_RECT, img\_id = 0,

color = 0, is\_screen\_img =

| button\_rect = (left = 24, top = 423, right = 73, bottom = 472)

| img\_rect = (left = 0, top = 0, right = 49, bottom = 49)

| lcd\_rect = (left = 24, top = 423, right = 73, bottom = 472)

| image\_display\_rect = (left = 24, top = 423, right = 73, bottom = 472)

| bg\_display = (is\_transparent = 0, is\_form\_bg = 0, rect = (left = 24, top

= 423, right = 73, bottom

-006|DisplayButton(

| button\_ctrl\_ptr = 0x045AF7CC,

| draw\_state = GUIBUTTON\_OWNDRAW\_NONE)

| cross\_rect = (left = 24, top = 423, right = 73, bottom = 472)

| button\_rect = (left = 0, top = 0, right = 0, bottom = 0)

| owner\_draw = (display\_x = 0, display\_y = 0, fg\_display\_x = 0,

fg\_display\_y = 0, display\_rect = (le

-007|ButtonHandleMsg(

| ?,

| ?,

| ?)

| result = 1

| button\_ctrl\_ptr = 0x045AF7CC

-008|MMK\_RunCtrlProc(

| ?,

| msg\_id = 61494,

| param = 0x0)

| me\_ptr = 0x045AF7CC

-009|MMK\_DispatchToHandle(

| handle = 0x37BA0079,

| msg\_id = 61494,

| param\_ptr = 0x0)

| bResult = 0

| ctrl\_handle\_result = 0

| type = MMI\_HANDLE\_CONTROL

| old\_handle = 16711680

-010|MMK\_SendMsg(

| ?,

| msg\_id = 61494,

| param\_ptr = 0x0)

| result = 0

-011|DrawCmdBtns(

| ?,

| ?,

| ?)

| screen\_width = 320

| screen\_height = 480

| img\_width = 50

| img\_height = 50

| rect = (left = 0, top = 7, right = 49, bottom = 70)

| bar\_width = 1

| bar\_height = 64

| bg\_width = 50

| bg\_height = 50

| win\_id = 1900547

| i = 1900570

-012|DisplayAllCmdBtns(

| ?)

| img\_left = 1075642421

| img\_mid = 0

| img\_right = 1075642422

-013|MMIDC\_DisplayAllCmdBtns(

| ?)

-014|MMIDC\_DisplaySoftKey(

| mode = DC\_REVIEW\_MODE)

-015|SetReviewOSDCallback()

| res = ((osd\_pos\_t = (x = 0, y = 0, w = 0, h = 0), next = 0x0),

(osd\_pos\_t = (x = 0, y = 0, w = 0,

-016|DCAMERA\_GetFrameFromIspNml(

| param\_ptr = 0x04685D8C,

| ?,

| ?)

| mem\_size = 614472

| dc\_dst\_size = (w = 640, h = 480)

| dc\_rect = (x = 0, y = 0, w = 640, h = 480)

-017|DCAMERA\_GetFrameFromISP(

| param\_ptr = 0x04685D8C,

| ?,

| bIsIspBypass = 0,

| quickview\_cb = 0x086F052F)

-018|DCAMERA\_DoSnapshot(

| param\_ptr = 0x04685D8C,

| return\_param\_ptr = 0x045DEED4)

| ret\_value = DCAMERA\_OP\_SUCCESS

| func\_ptr = 0x086F0C91

| select\_personal = 0

| is\_bypass\_isp = 0

| malloc\_buffer\_ptr = 0x0438C798

| malloc\_buffer\_len = 565168

| jpeg\_enc\_in\_param = (yuv\_addr = (y\_chn\_addr = 0, u\_chn\_addr = 0,

v\_chn\_addr = 0), image\_rect = (x

| jpeg\_enc\_out\_param = (out\_size = 0)

| call\_back\_return = 0

| temp\_yuv\_addr = (y\_chn\_addr = 0, u\_chn\_addr = 0, v\_chn\_addr = 0)

| lcd\_info = (r\_bitmask = 0, g\_bitmask = 0, b\_bitmask = 0, bits\_per\_pixel

= 0, lcd\_width = 0, lcd\_he

-019|DCAMERA\_TaskRoutine(

| ?,

| ?)

| signal = 0x045A874C

-020|ThreadEntry(

| ?)

| thread\_entry = (entry = 0x00047999, argc = 0, argv = 0x0)

-021|tx\_thread\_shell\_entry()

---|end of frame

t\_p\_app task的callback：

-000|tx\_thread\_suspend(

| ?)

-001|tx\_mutex\_get(

| ?,

| ?)

| thread\_ptr = 0x040350F4

-002|txe\_mutex\_get(

| ?,

| ?)

-003|SCI\_GetMutex(

| ?,

| ?)

| \_mutex\_ptr = 0x045991D4

-004|DCAMERA\_GetMutex(

| mutex = 0x045991D4)

-005|DCAMERA\_Close()

| eRet = DCAMERA\_OP\_SUCCESS

| lcd\_info = (r\_bitmask = 63488, g\_bitmask = 2016, b\_bitmask = 31,

bits\_per\_pixel = 16, lcd\_width = 320, lcd\_height = 4

-006|CloseDC()

-007|Exit()

-008|MMIDC\_FlowFunction(

| op = DC\_EXIT)

| res = 0

-009|ReleaseDCModule()

-010|DCApplet\_HandleEvent(

| ?,

| ?,

| ?)

| result = 1

-011|MMK\_RunAppletProc(

| ?,

| msg\_id = 61458,

| param = 0x0)

| me = 0x045A835C

-012|MMK\_DispatchToHandle(

| handle = 934019224,

| msg\_id = 61458,

| param\_ptr = 0x0)

| bResult = 0

-013|MMK\_SendMsg(

| ?,

| msg\_id = 61458,

| param\_ptr = 0x0)

| result = 0

-014|AppletTreeNodeDestruct(

| ?)

| result = 0

| applet\_node\_ptr = 0x0459F1EC

-015|MMK\_RemoveTreeNode(

| ?,

| func = 0x089AB9DB,

| ?)

-016|ReturnFirstApplet()

| i = 0

| applet\_system\_ptr = 0x045DB690

| travel\_ptr = 0x045A89E4

-017|MMK\_ReturnIdleWin()

-018|MMK\_HandleRedKeyProcess()

-019|MMK\_HandlePublicKey(

| msg\_id = 64280,

| ?)

-020|HandleMSGKbd(

| ?,

| ?)

| multi\_key\_tp\_param = (is\_slide = 0, pre\_tp\_point = (x = 0, y = 0),

cur\_tp\_point = (x = 0, y = 0), multi\_key\_num = 0,

-021|MMK\_DispatchMSGKbd(

| ?)

| key\_code = 24

| is\_long\_press = 0

-022|MMK\_DispatchExtSig(

| ?,

| ?)

-023|thread\_entry\_P\_APP(

| ?,

| ?)

| receiveSignal = 0x045CEF24

| mmi\_msg = 0x0456E74C

| ticks1 = 0

| ticks2 = 0

| is\_log\_on = 0

| watchdog\_ptr = 0x04596574

-024|ThreadEntry(

| ?)

| thread\_entry = (entry = 0x0845614F, argc = 0, argv = 0x0)

-025|tx\_thread\_shell\_entry()

---|end of frame

|

# “推导callback，结合log分析”办法

（例如bug1963）

Exception at 0x00000000

ASSERT(Reserved exception handler !)

该问题callback和死机位置很容易推导出来，死在ClearFileSegmentInfo函数中的SCI\_MEMSET(ebook\_seg\_ptr->ucs2\_str\_ptr,0,((MMIEBOOK\_SEG\_BUF\_UNIT\_LEN + 1)\*sizeof(wchar)));

ebook\_seg\_ptr参数是s\_ebook\_show\_info全局变量的参数，直接查看到s\_ebook\_show\_info.next\_show\_txt\_ptr->ucs2\_str\_ptr为0

ClearFileSegmentInfo(s\_ebook\_show\_info.next\_show\_txt\_ptr);

-000|ClearFileSegmentInfo(ebook\_seg\_ptr = 0x0)

-001|SwitchAndMakeNextSegment()

-002|MMIEbook\_HandleTaskMessage(?, param = 0x040333A0)

-003|MMIEBOOK\_SendMsgtoTask(?, ?, ?)

-004|MMIEBOOK\_TurnLineOrPage(?)

-005|HandleTxtShowWinMsg(?, ?, ?)

-006|MMK\_RunWinProc(?, msg\_id = 64002, param = 0x040334A4)

-007|MMK\_DispatchToHandle(handle = 1392181424, msg\_id = 64002,

-008|MMK\_DispMsgToFocusWin(msg\_id = 64002, param\_ptr = 0x04033

-009|MMK\_DispMsgToWin(msg\_id = 64002, param\_ptr = 0x040334A4)

-010|HandleMSGKbd(keys\_status = 64000, key\_code = 2)

-011|MMK\_DispatchMSGKbd(?)

-012|MMK\_DispatchExtSig(?, ?)

-013|thread\_entry\_P\_APP(?, ?)

-014|ThreadEntry(?)

-015|tx\_thread\_shell\_entry()

---|end of frame

接下去除了要增加指针和memset地址非空的判断外，就是要找出什么样的流程（操作步骤）导致此异常？

经过简单分析，正在处理消息的窗口是MMIEBOOK\_TXT\_SHOW\_WIN\_ID窗口；另一个窗口MMIEBOOK\_ENTER\_TXT\_CONTENT\_WIN\_ID已经关闭（查看s\_zorder\_system中不存在），而MMIEBOOK\_ENTER\_TXT\_CONTENT\_WIN\_ID关闭的时候就会将s\_ebook\_show\_info.next\_show\_txt\_ptr内容清空，问题就归结为为什么会出现MMIEBOOK\_ENTER\_TXT\_CONTENT\_WIN\_ID先于MMIEBOOK\_TXT\_SHOW\_WIN\_ID窗口关闭（而且后者存在还能响应按键）。在分析了log后，我们找到了答案。至此，此问题彻底分析完成，并且能够模拟写文件出错的情形来复现问题。

“可以找到FreeEbook的trace

"MMIEBOOK\_TASK MMIEBOOK\_FreeEbookTaskRes enter"

并且在前面有下面trace

"MMK\_RunWinProc, win\_handle = 0x52fa005b, win\_id = 0x00090003, win\_name =

MMIEBOOK\_ENTER\_TXT\_CONTENT\_WIN\_ID, msg\_id = 0x0000f022"

说明closeMMIEBOOK\_ENTER\_TXT\_CONTENT\_WIN\_ID窗口的时候触发了FreeEbook，

此后在MMIEBOOK\_TXT\_SHOW\_WIN\_ID窗口按KEYDOWN\_DOWN按键进入了最后导致死机的过程。

进一步分析发现HandleTxtShowWinMsg的MSG\_LOSE\_FOCUS消息处理中，WriteEbookBookMarkInfo写文件出错会触发closeMMIEBOOK\_ENTER\_TXT\_CONTENT\_WIN\_ID窗口动作

case MSG\_LOSE\_FOCUS:

if(!is\_disenable\_txt\_func)

{

WriteEbookBookMarkInfo();

}

”

# “推导callback，参数值代入分析”办法

以bug2285为例。首先推导出最深的callback如下，

-000|MMIAPISEARCH\_HandleControlMsg(?, ?, ?)

-001|MMIAPIISTYLESEARCH\_HandleSearchWinMsg(win\_id = 1, ?, ?)

-002|MMIIDLE\_IStyleHandleMsg(win\_id = 1, msg\_id = 57370, param = 0x040331C

-003|IdleWin\_HandleMsg(win\_id = 1, msg\_id = 57370, param = 0x040331C8)

-004|MMK\_RunWinProc(?, msg\_id = 57370, param = 0x040331C8)

-005|MMK\_DispatchToHandle(?, ?, ?)

-006|MMK\_SendMsg(?, msg\_id = 57370, param\_ptr = 0x040331C8)

-007|GUICTRL\_SendNotifyEx(?, code = 57370, notify\_ptr = 0x040331C8)

-008|SendNeedItemDataMsg(list\_ptr = 0x04555BFC, item\_index = 15)

-009|DrawAllItems(list\_ptr = 0x04555BFC, lcd\_dev\_info\_ptr = 0x04033258)

-010|DisplayListCtrl(list\_ptr = 0x04555BFC, inner\_state = 31)

-011|GUILIST\_HandleMsg(?, ?, ?)

-012|MMK\_RunCtrlProc(?, msg\_id = 61494, param = 0x0)

-013|ControlTreeNodeHandleEvent(?, ?, ?)

-014|MMK\_DispatchToAllTreeNode(?, func = 0x0805D76B, msg\_id = 61494, param

-015|MMK\_ProcSpecialWinMsg(win\_handle = 16908289, ?, param = 0x0)

-016|MMK\_DispatchToHandle(handle = 16908289, msg\_id = 61477, param\_ptr = 0

-017|MMK\_SendMsg(?, msg\_id = 61477, param\_ptr = 0x0)

-018|HandleFormNotifyResize(resize\_rect = (left = 0, top = 0, right = 0, b

-019|FormHandleMsg(?, ?, param = 0x04033450)

-020|MMK\_RunCtrlProc(?, msg\_id = 1, param = 0x04033450)

-021|MMK\_DispatchToHandle(handle = 266731711, msg\_id = 1, param\_ptr = 0x04

-022|MMK\_SendMsg(?, msg\_id = 1, param\_ptr = 0x04033450)

-023|GUICTRL\_SendNotifyEx(?, code = 1, notify\_ptr = 0x04033450)

-024|DestroyImCtrl(?, ?)

-025|HandleEditLoseActive(?, edit\_ctrl\_ptr = 0x04564544)

-026|EditHandleMsg(?, msg\_id = 4, param = 0x04033520)

-027|MMK\_RunCtrlProc(?, msg\_id = 4, param = 0x04033520)

-028|MMK\_DispatchToHandle(handle = 266862642, msg\_id = 4, param\_ptr = 0x04

-029|MMK\_SendMsg(?, msg\_id = 4, param\_ptr = 0x04033520)

-030|GUICTRL\_SendNotifyEx(?, code = 4, notify\_ptr = 0x04033520)

-031|GUICTRL\_SendNotify(ctrl\_handle = 278724673, code = 4)

-032|ImHandleMsg(?, ?, ?)

-033|MMK\_RunCtrlProc(?, msg\_id = 61491, param = 0x0)

-034|MMK\_SetActiveCtrl(?, is\_need\_paint = 0)

-035|MMK\_SetAtvCtrlEx(?, ?, is\_need\_paint = 0)

-036|DispatchMSGTpDown(?, ?)

-037|MMK\_DispatchMSGTp(?)

-038|MMK\_DispatchExtSig(?, ?)

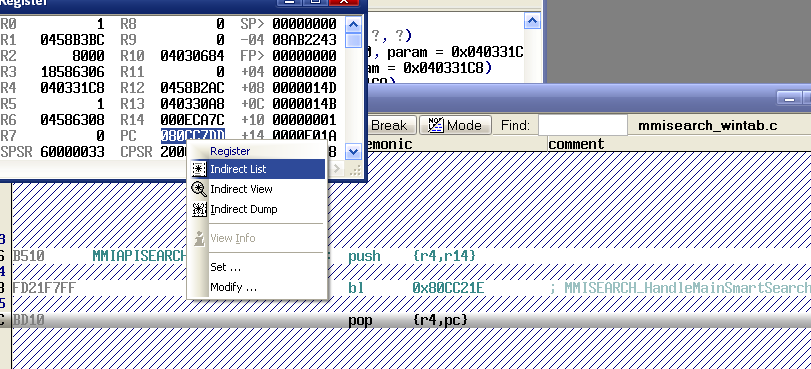
-039|thread\_entry\_P\_APP(?, ?)

-040|ThreadEntry(?)

-041|tx\_thread\_shell\_entry()

---|end of frame

接下去，可以查看pc的indirect list看到进入的函数，这里看到进入ST:080CC7D8FD21F7FF bl 0x80CC21E ; MMISEARCH\_HandleMainSmartSearchWinM



即进入了MMISEARCH\_HandleMainSmartSearchWinMsg函数。

接下去，在MMISEARCH\_HandleMainSmartSearchWinMsg函数发生了什么呢？这时候就需要分析出正确的参数值，并且代入推演了。

LOCAL MMI\_RESULT\_E MMISEARCH\_HandleMainSmartSearchWinMsg(

MMI\_WIN\_ID\_T win\_id,

MMI\_MESSAGE\_ID\_E msg\_id,

DPARAM param)

该函数有3个参数，可以通过寄存器压栈分析出实际的值，也可以通过查看上一次，上上次...函数调用的参数传递关系分析出实际的值。这里在MMIIDLE\_IstyleHandleMsg函数中可以看到这三个参数的实际值：win\_id = 1，msg\_id = 0xE01A，param = 0x040331C8。

-000|MMIAPISEARCH\_HandleControlMsg(

| ?,

| ?,

| ?)

| return = 1

-001|MMIAPIISTYLESEARCH\_HandleSearchWinMsg(

| win\_id = 1,

| ?,

| ?)

| result = 0

-002|MMIIDLE\_IStyleHandleMsg(

| win\_id = 1,

| msg\_id = 0xE01A,

| param = 0x040331C8)

-003|IdleWin\_HandleMsg(

MMISEARCH\_HandleMainSmartSearchWinMsg函数中有个很长的switch case，但是由于我们知道msg\_id = 0xE01A，所以我们只要直接跳过去分析case MSG\_CTL\_LIST\_NEED\_ITEM\_DATA:就行了。显然，又进入了SetSearchListItemData函数。

LOCAL MMI\_RESULT\_E MMISEARCH\_HandleMainSmartSearchWinMsg(

MMI\_WIN\_ID\_T win\_id,

MMI\_MESSAGE\_ID\_E msg\_id,

DPARAM param)

{

switch(msg\_id)

{

case MSG\_OPEN\_WINDOW:

...

case MSG\_CTL\_LIST\_NEED\_ITEM\_DATA:

SetSearchListItemData((GUILIST\_NEED\_ITEM\_DATA\_T\*)param);

break;

...

}

接下去将参数代入SetSearchListItemData函数继续推演。

LOCAL void SetSearchListItemData(GUILIST\_NEED\_ITEM\_DATA\_T \*need\_item\_data\_ptr )

{

MMISEARCH\_MATCH\_ITEM\_T \*match\_item\_ptr = PNULL;

uint16 pure\_list\_index = 0;

if(PNULL == need\_item\_data\_ptr)

{

return;

}

if (IsSplitItem(need\_item\_data\_ptr->item\_index))

{

SetSplitData(need\_item\_data\_ptr);

return ;

}

pure\_list\_index = DecSplitNum(need\_item\_data\_ptr->item\_index);

match\_item\_ptr = MMISEARCH\_GetOneMatchItemPtr(pure\_list\_index);

if (PNULL != match\_item\_ptr && PNULL != match\_item\_ptr->set\_data\_pfunc)

{

match\_item\_ptr->set\_data\_pfunc(need\_item\_data\_ptr, match\_item\_ptr->user\_data);

}

}

马上就到出现异常的地方了。

执行到MMISEARCH\_GetOneMatchItemPtr函数的时候出现问题，该函数的逻辑有点问题，导致返回无效地址（0+index）引起后续错误，

此时index=13, s\_search\_context.matched\_item\_num=2,

s\_search\_context.matched\_item\_ptr = PNULL。

PUBLIC MMISEARCH\_MATCH\_ITEM\_T\* MMISEARCH\_GetOneMatchItemPtr(uint32 index)

{

if ( index > s\_search\_context.matched\_item\_num && PNULL !=

s\_search\_context.matched\_item\_ptr)

{

return PNULL;

}

else

{

return &s\_search\_context.matched\_item\_ptr[index];

}

}

推导callback，参数和分析exception过程是经常能够用到的一个方法。有时候callback已经展开到最后一步了，那么直接就能看到死机的位置；另一些时候callback只能展开到倒数一步，就需要分析出实际的参数再代入推演，试试看吧，只要函数不是很复杂，少许有些耐心就能推演到实际死机的位置。

# “版本对比”办法

3月底到4月初报的很多问题（bug1511，bug1558，bug1575，bug1595…），经过一番分析后发现共同的特征“一片内存段高位04被改写成02，高位08被改写成了05”， 该问题属于不太好分析的内存覆盖问题，而且又不是必现，一时无法解决。随着测试的进行，类似问题不断爆出来。看样子，该问题用常规办法无法解决，此时thomas.chai采用了版本对比的办法，在没问题的A版本到出问题的B版本之间取若干个版本下载验证（有问题的版本平均测试20分钟就会出现该问题），另一方面借助测试的monkeytest数据，查找第一次出现此类问题的版本，直到定位到最终的引入版本revision。最终定位出来是在3-29的Revision7533上引入的，定位出来以后在4-5的Revision8310上解决了该问题。

下面是问题代码。在步骤1中mask\_rect计算出的是相对于lcd的坐标，而步骤2计算要用到的mask\_rect应该是相对于图层的坐标。在图层不是全屏的时候，这2者不一致就会导致步骤2执行完后bk\_buf\_ptr指向了一个异常地址，并且alpha减弱的算法造成该地址的内存块出现“高位04被改写成02，高位08被改写成05”的奇异现象。

修改以后的代码做了“相对于lcd的坐标”到“相对于图层的坐标”的转换，这个问题就没有了。

LOCAL void FillLayerWithMaskColor(GUI\_LCD\_DEV\_INFO \*layer\_ptr, GUI\_RECT\_T \*rect\_ptr, uint32 mask\_color)

{

UILAYER\_INFO\_T layer\_info = {0};

uint8 \*layer\_buf\_ptr = PNULL;

uint32 \*bk\_buf\_ptr = PNULL;

GUI\_RECT\_T mask\_rect = {0};

GUI\_RECT\_T layer\_rect = {0};

uint32 mask\_width = 0;

uint32 mask\_height = 0;

uint32 color = 0;

uint32 i = 0;

uint32 j = 0;

uint32 mask\_a = 0;

uint32 mask\_r = 0;

uint32 mask\_g = 0;

uint32 mask\_b = 0;

uint8 a = 0;

uint32 r = 0;

uint32 g = 0;

uint32 b = 0;

UILAYER\_GetLayerInfo(layer\_ptr, &layer\_info);

layer\_buf\_ptr = UILAYER\_GetLayerBufferPtr(layer\_ptr);

if (PNULL == layer\_buf\_ptr/\*layer\_info.layer\_buf\_ptr\*/)

{

return;

}

//步骤1

if (PNULL == rect\_ptr)

{

mask\_rect.right = layer\_info.layer\_width - 1;

mask\_rect.bottom = layer\_info.layer\_height - 1;

mask\_height = mask\_rect.bottom - mask\_rect.top + 1;

mask\_width = mask\_rect.right - mask\_rect.left + 1;

}

else

{

layer\_rect = UILAYER\_GetLayerRect(layer\_ptr);

if (!GUI\_IntersectRect(&mask\_rect, \*rect\_ptr, layer\_rect))

{

return;

}

mask\_height = mask\_rect.bottom - mask\_rect.top ;

mask\_width = mask\_rect.right - mask\_rect.left ;

}

mask\_a = (mask\_color>>24) & 0xFF;

mask\_r = (mask\_color>>16) & 0xFF;

mask\_g = (mask\_color>>8) & 0xFF;

mask\_b = (mask\_color) & 0xFF;

//步骤2

bk\_buf\_ptr = ((uint32\*)layer\_buf\_ptr/\*layer\_info.layer\_buf\_ptr\*/) + mask\_rect.top \* layer\_info.mem\_width + mask\_rect.left;

for (i = 0; i < mask\_height; i++)

{

for (j = 0; j < mask\_width; j++)

{

color = \*(bk\_buf\_ptr + j);

a = (color>>24) & 0xFF;

r = (color>>16) & 0xFF;

g = (color>>8) & 0xFF;

b = (color) & 0xFF;

color = ((((a \* 3 + mask\_a \* 5) >>3)<< 24) & 0xFF000000)

| ((((r \* 3 + mask\_r \* 5 ) >>3)<< 16) & 0x00FF0000)

| ((((g \* 3 + mask\_g \* 5 ) >>3) << 8) & 0x0000FF00)

| ((((b \* 3 + mask\_b \* 5)) >>3) & 0x000000FF);

\*(bk\_buf\_ptr + j) = color;

}

bk\_buf\_ptr += layer\_info.mem\_width;

}

}

更一般的，有时候对于一些新冒出来的问题，已经定位到某个文件出问题，但是还不能确定最后的原因，就会自然的查看一下该文件或者相关.c和.h文件的版本记录（show log），这其实就是用到了“版本对比”办法。

# 其他办法

## 查看嫌疑代码（文件）最近的修改记录（SVN：Show Log）

查看嫌疑代码（文件）最近的修改记录（SVN：Show Log）是最常用的辅助方法，可以穿插在其他办法中使用。

## 借助模拟器模拟和分析

## 必现问题上trace32上调试

## 必现内存覆盖问题调试办法

在覆盖嫌疑代码前后加MMI\_CheckAllocatedMemInfo(\_D\_FILE\_NAME, \_D\_FILE\_LINE);

以及通过busmonitor检测（后面有介绍）都是很常用的调试内存覆盖问题的方法。

## 查黑屏\花屏等问题的办法（源自james.zhang）

查黑屏\花屏等问题, 手动assert之后, 利用t32看图层的全局变量

利用t32工具的命令

//格式为 data.image 地址 宽 高 图片格式, 主层为565, 其他图层一般问888, 可根据layer结构中的type确定

data.image 0x45D410 320. 480. /RGB565LE

data.image 0x45D410 320. 440. /RGBX888LE

s\_layer\_blt 图层的叠加序列, 加入图层叠加序列后的图层, 才能显示

s\_layer\_arr 所有已经创建的图层, 展开后, 可看到所有的图层信息

以主层为例, 主层的block id为0, 起始buf\_addr为0x45b3700, 宽为320, 高为480, 格式为565, 输入命令后, 即可在新窗口中看到主层数据

好工具请大家用起来, 提高调试效率!

## MMI\_GetOneKey调试接口

已经添加了MMI\_GetOneKey调试接口（宏没有打开）：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Description : 等待按键，往下执行

// Global resource dependence :

// Author: bin.ji

// Note:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

PUBLIC void MMI\_GetOneKey(void);

以后大家调试界面显示的问题时，如果希望一帧帧的走，可以调用该接口，

步骤如下：

1）在mmimain.c文件中头上打开调试宏

#define MMI\_GETONEKEY\_DEBUG

2）在需要调试的代码中调用MMI\_GetOneKey接口，例如

PUBLIC void MMITHEME\_PlayZoomAnim(...)

{

...

for (; i <= in\_param.total\_frame; i++)

{

...

MMI\_GetOneKey();

}

}

就可以在程序运行到调试地方的时候停住，然后每按一次键走一帧，方便调试看现象了

## busmonitor检测方法

busmonitor检测方法可以用来检测某地址数据被异常修改。在现有6530版本，为使busmonitor正确起效，需要修改mem\_cfg.c文件，将rw data下面的VM\_CB\_WB改为0，作用是关闭写内存的cache（由于开启cache时，写内存数据会先进cache造成Busmonitor延迟或不准确），但是会使系统稍微变慢些。

//rw,ro region need align to 4kB(1pages)

//For every demand paging region pages need > 3 (page tyes:1k or 4k).

const static VM\_REGION vm\_region[] =

{

/\*--vir base addr-----size----- phy base addr--------type------access----control---- \*/

/\* rt code kernel img1, 4MB \*/

{/\*{0x00000000, 0x00400000, 0x00000000, VM\_USE\_RTCODE, 0, 0}\*/

MEM\_RO\_START\_ADDR,

MEM\_RO\_CODE\_SIZE,

MEM\_RO\_START\_ADDR,

VM\_USE\_RTCODE,

(VM\_PROT\_RW),

0

},

#ifdef MULTI\_BIN\_SUPPORT

/\* rt code kernel img1, 4MB \*/

{/\*{0x04000000, 0x04300000, 0x00000000, VM\_USE\_RTCODE, 0, 0}\*/

1. MEM\_RO2\_START\_ADDR,

MEM\_RO2\_CODE\_SIZE,

MEM\_RO2\_START\_ADDR,

VM\_USE\_RTCODE,

(VM\_PROT\_RW),

0

},

#endif

/\* rw data \*/

{

(MEM\_RW\_START\_ADDR),

(MEM\_RW\_AREA\_SIZE),

(MEM\_RW\_START\_ADDR),

(VM\_USE\_UNKNOWN),

(VM\_PROT\_RW),

(0/\*VM\_CB\_WB\*/)

},

接下去可以像下面代码这样调用busmonitor函数进行检测。

uint32 s\_bus\_data = 0;

uint32 \*s\_bus\_addr\_ptr = &s\_bus\_data;

void MonitorCallback(

unsigned long data,

unsigned long pc\_before\_fiq)

{

SCI\_ASSERT(0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Description :

// Global resource dependence :

// Author: bin.ji

// Note:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

PUBLIC void SetBusMonitor(void)

{

SCI\_TRACE\_LOW("SetBusMonitor() addr = 0x%x", s\_bus\_addr\_ptr);

bm\_monitor\_set(s\_bus\_addr\_ptr,

s\_bus\_addr\_ptr,

0xffffffff,

1/\*BM\_ACCESS\_MODE\_WRITE\*/,

MonitorCallback);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Description :

// Global resource dependence :

// Author: bin.ji

// Note:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

PUBLIC void TestBusMonitor(void)

{

SCI\_TRACE\_LOW("TestBusMonitor()");

s\_bus\_data = 1234;

}

0xffffffff可以用来检测任意值，只要往s\_bus\_addr\_ptr地址写任意数据都会进入MonitorCallback；如果填其余值，只有当变成该值的时候才会进入MonitorCallback。

上面代码可以用来验证busmoitor有效（系统起来以后调用一次SetBusMonitor，然后在某操作时调用TestBusMonitor，此时即会出现assert）。

下面是实际用busmonitor检测分析出的一个例子（busmonitor验证bug3842）：

直接用trace32cmm\_auto.bat推出的callback有点问题，对照ass文件手动推导出来的callback如下

r13:040349D0

pc:08D873AF代入得到下面callback

-000|DisplayHighlightIconLayer(

| ?)

| i = 0

| buf = 0x045DC1E0

| num = 0

| layer\_info\_ptr = (layer\_width = 320, layer\_height = 480, mem\_width = 320,

-001|DisplaySlidePageMenu(

| menu\_ctrl\_ptr = 0x045ABA6C)

| rect = (left = 0, top = 0, right = 319, bottom = 479)

-002|MenuHandleMsg(

| ?,

| ?,

| ?)

| result = 1

| menu\_ctrl\_ptr = 0x045ABA6C

-003|MMK\_RunCtrlProc(

| ?,

| msg\_id = 61494,

| param = 0x0)

| me\_ptr = 0x045ABA6C

-004|ControlTreeNodeHandleEvent(

| ?,

| ?,

| ?)

-005|MMK\_DispatchToAllTreeNode(

| ?,

| func = 0x083A030B,

| msg\_id = 61494,

| param = 0x0,

| ?)

| i = 0

| travel\_ptr = 0x045A99C4

-006|MMK\_ProcSpecialWinMsg(

| win\_handle = 20054064,

| ?,

| param = 0x0)

| result = 1

| win\_node\_ptr = 0x045ABA0C

| lcd\_dev\_info = (lcd\_id = 0, block\_id = 0)

-007|MMK\_DispatchToHandle(

| handle = 20054064,

| msg\_id = 61477,

| param\_ptr = 0x0)

| def\_handle\_result = 0

| openwin\_handle\_result = 1

| win\_handle\_result = 1

| old\_handle = 20447286

-008|MMK\_SendMsg(

| ?,

| msg\_id = 61477,

| param\_ptr = 0x0)

| result = 0

-009|MMK\_UpdateScreen()

| zorder\_system\_ptr = 0x04634120

| handle\_array = (20054064, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

| i = 0

| num = 1

-010|MMK\_ProcSpecialAppletMsg(

| ?,

| ?,

| ?)

| result = 0

-011|MMK\_DispatchToHandle(

| handle = 16711680,

| msg\_id = 61456,

| param\_ptr = 0x0)

| bResult = 0

-012|MMK\_DispatchWinMSG(

| mmi\_msg\_ptr = 0x045769DC)

-013|WRE\_GetMessageExx(

| lpMsg = 0x046B3DEC,

| ?,

| ?,

| ?)

| mmi\_msg = 0x045769DC

| needrefresh = 0

| receiveSignal = 0x0

| watchdog\_ptr = 0x0459E954

-014|GetMessageExx(

| ?,

| ?,

| ?,

| ?)

-015|StartDisktopServerNew(

| hInstance = 0xFFFFFFFF,

| ?,

| ?,

| ?)

-016|WKERNEL\_MwinWin\_HandleMsg(

| win\_id = 3342337,

| ?,

| ?)

| recode = 1

| Msg = (link = (next = 0x0, prev = 0x0), hwnd = 0x0, message = 0, wParam =

| gEngineeringKeyIndex = -1

| ret = 1

-017|WreMwinWin\_HandleMsg(

| win\_id = 3342337,

| msg\_id = 2147483648,

| param = 0x0)

| b\_need\_fill = 1

| mwin\_msg\_id = 12288

-018|HandleMsg(

| win\_id = 3342337,

| msg\_id = 2147483648,

| param = 0x0)

| recode = 150328643

-019|MMK\_RunWinProc(

| ?,

| msg\_id = 2147483648,

| param = 0x0)

| win\_id\_name = 0x04605004

-020|MMK\_DispatchToHandle(

| handle = 20447286,

| msg\_id = 2147483648,

| param\_ptr = 0x0)

| openwin\_handle\_result = 0

| type = MMI\_HANDLE\_WINDOW

| old\_handle = 16711680

-021|MMK\_DispatchWinMSG(

| mmi\_msg\_ptr = 0x045769CC)

-022|MMK\_DispatchMSGQueue(

| ?)

-023|thread\_entry\_P\_APP(

| ?,

| ?)

| receiveSignal = 0x0

| mmi\_msg = 0x045769CC

| ticks1 = 0

| ticks2 = 0

| is\_log\_on = 0

| watchdog\_ptr = 0x0459E954

-024|ThreadEntry(

| ?)

| thread\_entry = (entry = 0x083938DB, argc = 0, argv = 0x0)

-025|tx\_thread\_shell\_entry()

---|end of frame

查看主层s\_layer\_arr[0]数据结尾（0x44F9F1C），此时刚好被改写了。进入DisplayHighlightIconLayer查看相应的代码，很快就能分析出原因

从主菜单进入wre失败返回必现，

mmimenu\_slide\_page\_go.c文件的下面函数有问题，此时menu\_ctrl\_ptr->highlight\_layer\_handle={0,0}，操作的是主层，由于主层是565格式，以uint32填充必然导致内存覆盖。

LOCAL void DisplayHighlightIconLayer(GUIMENU\_CTRL\_T \*menu\_ctrl\_ptr)

{

#ifndef MMI\_PDA\_SUPPORT

#else

register uint32 i = 0;

uint32 \*OutBuf =

(uint32\*)UILAYER\_GetLayerBufferPtr(&menu\_ctrl\_ptr->highlight\_layer\_handle);

// IMGREF\_SIZE\_T size = {0};

register uint32 \*buf = OutBuf;

register uint32 num = 0;

register uint32 color = 0x00ffff00;//??é?

UILAYER\_INFO\_T layer\_info\_ptr = {0};

if(PNULL == OutBuf)

{

return;

}

...

while( i < num )

{

\*buf = color;//??é?

i++;

buf++;

}

#endif

}

## 寻找必现路径

很多时候，由于代码逻辑的复杂性，仅仅采用分析法从callback和死机现场进行分析很难分析出异常原因（有时候只是分析出被覆盖等直接原因），此时通过分析callback和Log等信息去寻找问题发生的必现路径往往可以得到很好的效果。

上面的例子bug3842就是通过分析callback猜测必现路径是主菜单操作wre返回发现必现路径，再采用busmonitor最终定位问题的。另外，还有bug3944也是如此。可以同时在模拟器上和手机上测试寻找必现路径。

还有很多时候，从测试哪里报过来是必现问题，而在研发这里验证却是非必现（或无法重现）问题。这样的矛盾在bugfix中非常常见，这其实是非常重要的信息，不能够忽略。

出现矛盾的原因是“测试环境或测试路径不同等影响因素造成测试结果不同”，例如bug3481就是插入有效SIM卡才能重现的必现问题。

常见的影响因素：

1）手机硬件因素

2）不插SIM卡/插入无效SIM卡/有效SIM卡

3）不插T卡/插T卡

4）不插耳机/插耳机

5）切换不同的系统（默认系统/智能系统/魔法桌面/三星系统）

6）执行路径不同（例如从主菜单进入，和从idle界面进入相同菜单，执行路径也是不同的）

同一台手机下载相同版本，零初始环境（不插耳机，不插SIM卡，不插T卡，第一次开机后执行相同路径）的情况下，运行结果一定是相同的。不同手机由于硬件的区别，有可能会有不同的运行结果。

同一台手机下载相同版本，在相同的非零初始环境（同样不插入（或插入）耳机，插入相同SIM卡，插入相同T卡，第一次开机后执行相同路径）的情况下，一般来说运行结果也是相同的。不同手机由于硬件的区别，有可能会有不同的运行结果。

因此，当出现“重现矛盾”时（非必现问题也是“重现矛盾”的一种），就要耐心的找出产生矛盾的影响因素，找到必现路径，接下去就能着手解决了。

# Bug3341

Bug3341是个很有趣的例子，对此问题发生过程的分析可以揭示内存覆盖问题可以怎样引起callback断掉，怎样引起貌似Unexpectly PC set to 0普通问题的完整过程。另外，此问题的解决过程综合了trace32调试和版本对比的方法。

先说一下解决过程，由于该问题在当天晚上的monkeytest出现的频率很高，经过callback的初步分析发现了必现步骤：“下拉状态栏，点击最后一个shortcur\_item项，开启/关闭，必现死机”，既然发现了必现步骤当然可以上trace32调试。其次该问题是当天引入的，前一天还没有，再结合高出现频率，于是去[\\shpld03\build\_result](file:///\\shpld03\build_result)取主干最近的版本，二分法查找引入问题的那次修改的revision。

下面详细分析一下此问题的发生过程。

首先，发现最后的callback是断掉的，

-000|TXAS\_SystemAssert(

| exp = 0x06B0,

| file = 0x06A4,

| line = 144,

| assert\_info\_ptr = 0x045F6499)

| assert\_mode = 1

| cur\_sp = 67322456

| cur\_lr = 4853

| cur\_pc = 13268

| i = 0

-001|SCI\_PAssert(

| ?,

| ?,

| ?,

| ?)

-002|\_\_PureAssert()

-003|SCI\_RestoreIRQ()

---|end of frame

然后可以用前面提到的方法推出从thread入口开始的callback，

r13:040342E0

pc:08933913代入可得下面callback

-000|HandleDropDownTPUp(

| win\_id = 3145729,

| tp\_point\_ptr = 0x045E1484)

| lcd\_width = 480

| lcd\_height = 320

| i = 0

-001|HandleDropDownWinMsg(

| win\_id = 3145729,

| ?,

| ?)

| s\_is\_date\_on = 0

| result = 1

| point = (x = 419, y = 48)

| lcd\_dev\_info = (lcd\_id = 0, block\_id = 0)

-002|MMK\_RunWinProc(

| ?,

| msg\_id = 61442,

| param = 0x0403448C)

| win\_id\_name = 0x0461141C

-003|DispatchMSGTpUp(

| ?,

| ?)

| win\_handle = 160825436

| point = (x = 419, y = 48)

| tp\_handle\_result = 0

| para = (is\_slide = 0, pre\_tp\_point = (x = 0, y = 0), cur\_tp\_point = (x =

| msg\_id = 61442

| old\_handle = 16711680

-004|MMK\_DispatchMSGTp(

| ?)

| tppress\_ptr = 0x045DF844

-005|MMK\_DispatchExtSig(

| ?,

| ?)

-006|thread\_entry\_P\_APP(

| ?,

| ?)

| receiveSignal = 0x045DF834

| mmi\_msg = 0x0459E464

| ticks1 = 0

| ticks2 = 0

| is\_log\_on = 0

| watchdog\_ptr = 0x045C63BC

-007|ThreadEntry(

| ?)

| thread\_entry = (entry = 0x0833CF1B, argc = 0, argv = 0x0)

-008|tx\_thread\_shell\_entry()

---|end of frame

接下去分析下一步进入了Shortcut\_SetSaveMode，根据该函数的push个数推断，r13将移到0x40342C4位置，并在该位置压入pc返回地址。但是该位置实际值是04674398不是一个函数地址，而根据\_\_PureAssert反推到的r13是0x40342C0，跟上述位置也有一定的偏移。这里存在着矛盾，该如何解释？可以猜测局部变量越界覆盖造成了0x40342C4位置的pc返回地址被改写，但如何解释\_\_PureAssert反推到的r13偏移？

trace32单步跟踪告诉了我们答案。

上次修改code合入错误造成了mmiset\_nv.c/mmiset\_nv.h的不匹配，导致MMINV\_SET\_POWER\_SAVING\_OPRATE\_ENV\_MODE该NV值的size错误，从而引发下面的代码出现上面猜测的mode\_id局部变量越界覆盖错误，造成了0x40342C4位置的pc返回地址被改写。

PUBLIC uint8 MMIENVSET\_GetPowerSavENVMode(void)

{

MN\_RETURN\_RESULT\_E return\_value = MN\_RETURN\_FAILURE;

uint8 mode\_id = 0;

MMINV\_READ(MMINV\_SET\_POWER\_SAVING\_OPRATE\_ENV\_MODE, &mode\_id, return\_value);

if (MN\_RETURN\_SUCCESS != return\_value)

{

mode\_id = 0;

MMINV\_WRITE(MMINV\_SET\_POWER\_SAVING\_OPRATE\_ENV\_MODE, &mode\_id);

}

return mode\_id;

}

通常覆盖会有3种情况：覆盖为0，覆盖成数据，覆盖为函数地址。覆盖为0，pc执行会出现通常的Unexpectly PC set to 0，但是r13不会偏移，解释不了目前的情况；覆盖成数据，会出现alignment错误或者undefine错误，也解释不了目前的情况；只能是覆盖成函数地址，但是这难道是巧合吗？

进一步通过trace32跟踪以及分析代码，可以发现这样一个过程，

“函数的调用过程如下：HandleDropDownTPUp->Shortcut\_SetSaveMode->MMIAPIENVSET\_ResetActModeOtherRingSet->MMIENVSET\_GetPowerSavENVMode

其中MMIENVSET\_GetPowerSavENVMode会先读后写NV，MMIAPIENVSET\_ResetActModeOtherRingSet会再次写该NV。0x836c291是位于MMIAPIENVSET\_ResetActModeOtherRingSet

的一个地址，也是MMIENVSET\_GetPowerSavENVMode返回的地址，同理，0x837fcdb是位于Shortcut\_SetSaveMode的一个地址，也是MMIAPIENVSET\_ResetActModeOtherRingSet

返回的地址，0x837f84b是位于HandleDropDownTPUp中的一个地址，也是Shortcut\_SetSaveMode的返回地址。下面演示了如何从MMIENVSET\_GetPowerSavENVMode直接返回到

Shortcut\_SetSaveMode中，而从Shortcut\_SetSaveMode返回时由于弹出的PC不是原来压栈的PC造成了PC跳到0的过程。

没写NV之前

sp:0x4033688 -> 00000000 Flash:0x1FE92DE -> FF FF FF FF

00000000 FF FF FF FF

04033804 FF FF FF FF

return addr 08362C91 FF FF FF FF

00000000

00000000

04033804

return addr 0837FCDB

00000000

00000000

00000000

00000000

00000003

return addr 0837f84B

........

第一次写NV之后

0x4033688 -> 00000000 Flash:0x1FE92DE -> 00 00 00 00

00000000 00 00 00 00

04033804 04 38 03 04

return addr 08362C91 91 2C 36 08

sp:0x4033698 -> 00000000

00000000

04033804

return addr 0837FCDB

00000000

00000000

00000000

00000000

00000003

return addr 0837f84B

........

第二次写NV之后

0x4033688 -> 00000000 Flash:0x1FE92DE -> 00 00 00 00

00000000 00 00 00 00

04033804 04 38 03 04

return addr 08362C91 DB FC 37 08

sp:0x4033698 -> 00000000

00000000

04033804

return addr 0837FCDB

00000000

00000000

00000000

00000000

00000003

return addr 0837f84B

........

下次读NV之前

sp:0x4033688 -> 00000000 Flash:0x1FE92DE -> 00 00 00 00

00000000 00 00 00 00

04033804 04 38 03 04

return addr 08362C91 DB FC 37 08

00000000

00000000

04033804

return addr 0837FCDB

00000000

00000000

00000000

00000000

00000003

return addr 0837f84B

........

下次读NV之后

sp:0x4033688 -> 00000000 Flash:0x1FE92DE -> 00 00 00 00

00000000 00 00 00 00

04033804 04 38 03 04

return addr 0837FCDB DB FC 37 08

00000000

00000000

04033804

return addr 0837FCDB

00000000

00000000

00000000

00000000

00000003

return addr 0837f84B

........”

了解这个过程就能知道这不是巧合，而是代码执行导致的。我们可以看到，错综复杂的软件代码执行过程，由于中间的一点错误（例如内存覆盖），就可能导致很多奇奇怪怪的问题，但是只要重现该执行过程，将该过程走一遍，就一点也不奇怪。

# Bug4967

Bug4967也是一个内存覆盖引起callback断掉的问题。分析这个问题可以看出常见的内存覆盖引起callback断掉的一点端倪。

假如调用关系为A->X->B，那么X函数内部发生的栈内存覆盖，会导致callback断裂为…->A，X->B…两段（栈内存往增大的方向覆盖，会把A调用X的返回地址给覆盖掉）。因此“死机时的callback”的断裂段的底部的那个函数内部发生“栈内存覆盖”的嫌疑最大，这个例子中是InitPdaCallOtherSettingsCtrl函数。其次，最接近X调用B的位置附近代码的嫌疑最大（一旦覆盖，通常马上会引起问题），这个例子中是调用AppendPdaSettingsItem2Line前的default处理。

通常还可以辅助看看嫌疑代码（文件）最近的修改记录（SVN：Show Log）。

default :

{

wchar temp\_wchar[3 + 1]={0};

char temp\_char[2 + 1] = {0};

MMI\_STRING\_T second\_str = {0};

text\_str.wstr\_ptr = temp\_wchar;

sprintf((char\*)temp\_char, "%d", MMIAPISET\_GetMinuteReminderTime());

text\_str.wstr\_len = strlen((char\*)temp\_char);

MMI\_STRNTOWSTR(text\_str.wstr\_ptr, text\_str.wstr\_len, (uint8 \*)temp\_char, text\_str.wstr\_len, text\_str.wstr\_len);

MMI\_GetLabelTextByLang(TXT\_SECOND, &second\_str);

text\_str.wstr\_len += second\_str.wstr\_len;

MMIAPICOM\_Wstrcat(text\_str.wstr\_ptr, second\_str.wstr\_ptr);

}

break;

}

AppendPdaSettingsItem2Line(ctrl\_id,TXT\_SET\_CALL\_TIME\_REMINDER,text\_str,IMAGE\_NULL,MMISET\_CALL\_MINUTE\_REMINDER\_LABEL\_EDIT\_CTRL\_ID);

断掉前后的callback分别如下，

死机时的callback：

-000|MMIAPICOM\_Wstrncpy(

| dst = 0x040A0DB8,

| src = 0x0E40002A,

| count = 3623)

| start = 0x040A0DB8

-001|AppendPdaSettingsItem2Line(

| ctrl\_id = 786857,

| text\_1 = 1074528318,

| text\_str = (wstr\_ptr = 0x0E40002A, wstr\_len = 3623),

| image\_id = 1073741824,

| user\_data = 787079)

| item\_t = (item\_style = 0, item\_group = 0, group\_idx = 0, first\_letter =

| item\_data = (item\_content = ((item\_data = (text\_id = 0, text\_buffer = (

| temp\_wstr = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

| temp\_len = 0

-002|InitPdaCallOtherSettingsCtrl()

| ctrl\_id = 786857

| text\_str = (wstr\_ptr = 0x0E40002A, wstr\_len = 3623)

| text\_str\_null = (wstr\_ptr = 0x0E170E32, wstr\_len = 3637)

| temp\_wchar = (49, 3623, 3636, 3609)

| temp\_char = (49, 0, 0)

| second\_str = (wstr\_ptr = 0x0883C1F0, wstr\_len = 6)

| mode\_info = (is\_sms\_reply\_open = 0, is\_anykey\_answer = 0)

-003|SR:0xE210E48(asm)

---|end of frame

|

|

r13:040A0F20

pc:00024A4B代入得到断掉前的callback，根据当前s\_zorder\_system推得缺少的函数是HandleSetCallSetOtherWindow，

-000|MMK\_RunWinProc(

| ?,

| msg\_id = 0,

| param = 0x0)

| win\_id\_name = 0x042F6C60

-001|MMK\_DispatchToHandle(

| handle = 1787035692,

| msg\_id = 61473,

| param\_ptr = 0x0)

| openwin\_handle\_result = 1

| old\_handle = 16711680

-002|MMK\_DispatchWinMSG(

| mmi\_msg\_ptr = 0x04291944)

-003|MMK\_DispatchMSGQueue(

| ?)

-004|thread\_entry\_P\_APP(

| ?,

| ?)

| receiveSignal = 0x0

| mmi\_msg = 0x04291944

| ticks1 = 0

| ticks2 = 0

| is\_log\_on = 0

| watchdog\_ptr = 0x042924FC

-005|ThreadEntry(

| ?)

| thread\_entry = (entry = 0x000199ED, argc = 0, argv = 0x0)

-006|tx\_thread\_shell\_entry()

---|end of frame

|

# debug中常用的数据结构，全局变量和trace

S\_static\_thread\_table：各task的信息（T\_p\_app task的thread\_id=21）

S\_zorder\_system：MMI窗口管理链表

S\_handle\_list：MMI窗口控件和applet管理链表（s\_handle\_list[0x??]，handle的低16位填入0x??处）

S\_layer\_blt：图层叠加序列

S\_layer\_arr：图层信息数据结构

MEM\_HEADER\_T：内存头的数据结构

typedef struct MEM\_HEADER\_tag{

struct MEM\_HEADER\_tag \* pre;

struct MEM\_HEADER\_tag \* succ;

char \* file\_name;

int line;

unsigned long size;

unsigned long block\_num;

} MEM\_HEADER\_T;

形如MMK\_RunWinProc, win\_handle = 0x01390032, win\_id = 0x002b003c, win\_name = WIDGET\_WEATHER\_TWO\_WIN\_ID, msg\_id = 0x0000f025的trace

# Bug调试系统知识

* 1. T\_p\_app task总共只有12KB（SC6530\_sc6530\_64X64\_240X400PDA工程）。而局部变量是放在栈里面的，会占用栈size，所以要注意不能使用特别大的局部变量（改成动态分配）。

huitao大侠推荐了一个查看函数调用栈的方法，这里也给大家推荐一下，

《RVDS31连接器和实用工具使用指南.pdf》记载的--callgraph指令，可以在编译链接的时候输出函数调用栈的情况，有必要让大家扫描一下（因为目前t\_p\_app task的stack只有12KB左右，很多函数调用栈已经超过此限制，如果运行起来必定会出现死机）

修改编译文件makefile，增加下面红色的选项

-$(LINK) --entry \_\_vectors --callgraph --callgraph\_output=text --cgfile=user --map --symbols ...

编译链接就能够生成SC6530\_sc6530\_XXX.txt文件，该文件中记录了详细的函数栈使用信息，下面是2个例子，很明显这样的执行序列会导致stack深度达到13KB，超出了目前的t\_p\_app task的stack size。

**强烈建议：找出所有Max Depth>12KB的并修改，找出所有Stack size > 2KB的并修改**

1)

MMIBROWSER\_Add2MyNav (Thumb, 208 bytes, Stack size 4904 bytes, mmibrowser\_wintable.o(.text))

[Stack]

Max Depth = 13536 + Unknown Stack Size

Call Chain = MMIBROWSER\_Add2MyNav => MMIPUB\_OpenAlertWarningWin => MMIPUB\_OpenAlertWinByTextId => MMIPUB\_OpenAlertWinByTextIdEx => OpenAlertWin => CreatAnimLabelCtrl => GUIANIM\_SetParam => SetAndDecodeData => DecodeImgBufOrFile => HandleAnimGetDataCnf => DecodeImgBufOrFile (Cycle)

[Calls]

\* \_\_aeabi\_memclr4 (via BLX)

\* \_\_aeabi\_memcpy4 (via BLX)

\* SCI\_Malloc\_Zero (via BLX)

\* MMK\_CreateWin

\* MMIPUB\_OpenAlertWarningWin (via BLX)

\* MMIBROWSER\_GetMyNavInfo (via BLX)

\* GUIEDIT\_SetString (via BLX)

[Called By]

\* BrwHandleBookmarkPopUpWinMsg (via BLX)

\* BrwHandleBookmarkPopMenuWinMsg (via BLX)

\* BrwHandleHistoryPopUpWinMsg (via BLX)

\* BrwHandleHistoryMenuWinMsg (via BLX)

2)

MMISMS\_RecvVcardOrVcalSms (Thumb, 1550 bytes, Stack size 2560 bytes, mmisms\_receive.o(.text))

[Stack]

Max Depth = 13216 + Unknown Stack Size

Call Chain = MMISMS\_RecvVcardOrVcalSms => MMISMS\_PlayNewSMSRing => PlaySMSRing => MMIAPISET\_PlayRing => PlayRing => PlayFixedRing => MMIPUB\_OpenAlertWarningWin => MMIPUB\_OpenAlertWinByTextId => MMIPUB\_OpenAlertWinByTextIdEx => OpenAlertWin => CreatAnimLabelCtrl => GUIANIM\_SetParam => SetAndDecodeData => DecodeImgBufOrFile => HandleAnimGetDataCnf => DecodeImgBufOrFile (Cycle)

[Calls]

\* \_\_aeabi\_memclr4 (via BLX)

\* \_\_aeabi\_memcpy (via BLX)

\* \_\_1sprintf (via BLX)

\* SCI\_MallocApp\_Zero (via BLX)

\* SCI\_Free\_Debug (via BLX)

\* SCI\_Trace\_ID (via BLX)

\* MMIDEFAULT\_TurnOnBackLight (via BLX)

\* MMK\_CloseWin (via BLX)

\* MMK\_GetFirstAppletHandle (via BLX)

\* MMK\_IsOpenWin (via BLX)

\* TM\_GetTotalSeconds (via BLX)

\* MNSMS\_FreeUserDataHeadSpaceEx (via BLX)

\* MNSMS\_DecodeUserDataHeadEx (via BLX)

\* MMISMS\_SetNewSMSRingFlag

\* MMISMS\_PlayNewSMSRing

\* MMISMS\_SaveVcardSms

\* MMIVCAL\_SaveVCalendarByData

[Called By]

\* MMISMS\_OperateMtIndForVcardOrVcal

* 1. T\_p\_app task的thread\_id=21，priority=76。
  2. ================== System Space Information

Ass文件中该区域是大块内存分配情况。

* 1. ============= System Space Used PlayBack==============

Ass文件中该区域是大块内存分配释放的历史记录。

* 1. ------------App Space New Segment--------------------

Ass文件中该区域是大块内存sci\_mem.c分配的内存的具体信息。

# 附件（引用的例子）

NEWMS00199242

bug721（缺版本和log）

bug1035

bug1102

bug1321

bug1405

bug1511（缺版本和log）

bug1557

bug1558

bug1571（缺版本和log）

bug1575（缺版本和log）

bug1595（缺版本和log）

bug1806

bug1817

bug1916（缺版本）

bug1963（缺版本和log）

bug2006（缺版本和log）

Bug2285

bug2573

bug2708

bug2794