# linux内核驱动调试方法

## 1. printk

### 1.1 打印级别

在include/linux/kernel.h中，下面几个宏控制了printk输出信息的记录级别。

#define console\_loglevel (console\_printk[0])

#define default\_message\_loglevel (console\_printk[1])

#define minimum\_console\_loglevel (console\_printk[2])

#define default\_console\_loglevel (console\_printk[3])

int console\_printk**[**4**]** **=** **{**

DEFAULT\_CONSOLE\_LOGLEVEL**,** /\* console\_loglevel \*/

DEFAULT\_MESSAGE\_LOGLEVEL**,** /\* default\_message\_loglevel \*/

MINIMUM\_CONSOLE\_LOGLEVEL**,** /\* minimum\_console\_loglevel \*/

DEFAULT\_CONSOLE\_LOGLEVEL**,** /\* default\_console\_loglevel \*/

**};**

#define DEFAULT\_MESSAGE\_LOGLEVEL 4 /\* KERN\_WARNING \*/

#define MINIMUM\_CONSOLE\_LOGLEVEL 1 /\* Minimum loglevel we let people use \*/

#define DEFAULT\_CONSOLE\_LOGLEVEL 7 /\* anything MORE serious than KERN\_DEBUG \*/

(1)console\_loglevel为7，代表printk打印级别小于7，这个信息才会被打印。  
(2)default\_message\_loglevel为4，代表printk的参数不加打印级别的时候的默认打印级别，默认为KERN\_WARNING.  
(3)minimum\_console\_loglevel是个预设值，平时不起作用，通过其他工具来设置console\_loglevel时，这个值不能小于minimum\_console\_loglevel。这里设置为1，也就是说不能设置0,-1之类的console\_loglevel等级。  
(4)default\_console\_loglevel是个预设值，平时不起作用，它表示设置console\_loglevel时的默认值。

可以通过修改/proc/sys/kernel/printk来改变上述的打印等级设置。

# cat /proc/sys/kernel/printk

7 4 1 7

#echo "1 4 1 7" > /proc/sys/kernel/printk

console\_loglevel设置为1时，所有的printk信息都不会被打印。

调用printk时的打印级别有：

#define KERN\_EMERG "<0>" /\* system is unusable \*/

#define KERN\_ALERT "<1>" /\* action must be taken immediately \*/

#define KERN\_CRIT "<2>" /\* critical conditions \*/

#define KERN\_ERR "<3>" /\* error conditions \*/

#define KERN\_WARNING "<4>" /\* warning conditions \*/

#define KERN\_NOTICE "<5>" /\* normal but significant condition \*/

#define KERN\_INFO "<6>" /\* informational \*/

#define KERN\_DEBUG "<7>" /\* debug-level messages \*/

### 1.2 printk和串口的关系

printk调用路径如下：

printk

vprintk**(**fmt**,** args**);**

vscnprintf**(**printk\_buf**,** **sizeof(**printk\_buf**),** fmt**,** args**);**

vsnprintf**(**buf**,**size**,**fmt**,**args**);**//先把输出信息输入到临时buffer

release\_console\_sem**();**

call\_console\_drivers**(**\_con\_start**,** \_log\_end**);**

\_call\_console\_drivers**(**start\_print**,** cur\_index**,** msg\_level**);**

\_\_call\_console\_drivers**(**start**,** end**);**

con**->**write**(**con**,** **&**LOG\_BUF**(**start**),** end **-** start**);**//调用具体的输出函数

2440会调用下面的函数进行注册：

console\_initcall**(**s3c24xx\_serial\_initconsole**);**

s3c24xx\_serial\_initconsole

     register\_console**(&**s3c24xx\_serial\_console**);**

s3c24xx\_serial\_console结构体内容如下：

static struct console s3c24xx\_serial\_console **=**

**{**

**.**name **=** S3C24XX\_SERIAL\_NAME**,**

**.**device **=** uart\_console\_device**,**

**.**flags **=** CON\_PRINTBUFFER**,**

**.**index **=** **-**1**,**

**.**write **=** s3c24xx\_serial\_console\_write**,**

**.**setup **=** s3c24xx\_serial\_console\_setup

**};**

static void

s3c24xx\_serial\_console\_write**(**struct console **\***co**,** const char **\***s**,**

unsigned int count**)**

**{**

uart\_console\_write**(**cons\_uart**,** s**,** count**,** s3c24xx\_serial\_console\_putchar**);**

**}**

最终会调用uart\_console\_write输出字符到串口。

### 1.3 ttySAC0与串口的关系

uboot会传递console=ttySAC0的参数进来，printk是如何寻找到该设备呢？  
解析uboot的console参数是通过\_\_setup("console=", console\_setup);注册的。  
也就是会调用console\_setup去解析console=ttySAC0，它会解析出console设备名为ttySAC，索引号为0。这些信息保存在console\_cmdline中。

struct console\_cmdline

**{**

char name**[**8**];** /\* Name of the driver \*/

int index**;** /\* Minor dev. to use \*/

char **\***options**;** /\* Options for the driver \*/

**};**

#define MAX\_CMDLINECONSOLES 8

static struct console\_cmdline console\_cmdline**[**MAX\_CMDLINECONSOLES**];**

在前面注册函数中s3c24xx\_serial\_console的名称为ttySAC，索引为-1。-1表示使用命令行中解析出来的索引，表示串口0。

## 2. OOPS

### 2.1 OOPS文件分析

Oops这个单词含义为“惊讶”当内核出错时(比如访问非法地址)打印出来的信息被称为 Oops信息。

我们先来人为的产生一个异常。

static int first\_drv\_init**(**void**)**

**{**

…………

int **\***p **=** 1**;**

**\***p **=** 1**;**

**return** 0**;**

**}**

运行的时候会出现oops信息，下面来逐一分析。  
首先是一段文本描述信息，说明发生了哪类的错误。这里表示访问0x01地址的时候出现了非法访问。

Unable to handle kernel **NULL** pointer dereference at virtual address 00000001

接下来打印虚拟内存页目录，页表信息，oops错误号，内核中加载的模块名称，发生错误的cpu序号等信息。

pgd **=** c3db0000

**[**00000001**]** **\***pgd**=**33d7c031**,** **\***pte**=**00000000**,** **\***ppte**=**00000000

Internal error**:** Oops**:** 813 **[**#1**]**

Modules linked in**:** led\_drv

CPU**:** 0 Not tainted **(**2.6.22.6 #3**)**

PC值告诉我们出错时的位置，pc位于first\_drv\_init后的0x94处。等下会展示如何寻找到该位置，另外还有一个0xcc，代表函数总长度。

PC is at first\_drv\_init**+**0x94**/**0xcc **[**led\_drv**]**

LR is at 0xc486e050

接下来会输出一些寄存器信息，并指出了发生错时当前进程的名称是insmod。

pc **:** **[<**bf000154**>]** lr **:** **[<**c486e050**>]** psr**:** 00000013

sp **:** c3d4bec0 ip **:** 00000000 fp **:** c3d4bed4

r10**:** c486e000 r9 **:** c06fc140 r8 **:** 0000001a

r7 **:** c487a260 r6 **:** bf000840 r5 **:** bf000840 r4 **:** bf0009c8

r3 **:** bf0009c4 r2 **:** c486e054 r1 **:** 00000001 r0 **:** 00000000

Flags**:** nzcv IRQs on FIQs on Mode SVC\_32 Segment user

Control**:** c000717f Table**:** 33db0000 DAC**:** 00000015

Process insmod **(**pid**:** 770**,** stack limit **=** 0xc3d4a258**)**

backstrace表示回溯信息，告诉调试者在oops出错前，模块调用了哪些函数。从下面信息看first\_drv\_init函数出错了。  
这段信息分为两部分，表示sys\_init\_module调用了first\_drv\_init函数。  
bf0000c0是first\_drv\_init函数首地址偏移0的地址，这个函数大小为0xcc。  
c006385c是sys\_init\_module函数首地址偏移0x1424的地址，这个函数大小为0x1514。

Backtrace**:**

**[<**bf0000c0**>]** **(**first\_drv\_init**+**0x0**/**0xcc **[**led\_drv**])** from **[<**c006385c**>]** **(**sys\_init\_module**+**0x1424**/**0x1514**)**

r4**:**00000000

**[<**c0062438**>]** **(**sys\_init\_module**+**0x0**/**0x1514**)** from **[<**c002bea0**>]** **(**ret\_fast\_syscall**+**0x0**/**0x2c**)**

Code表示出错指令附近的机器码，出错指令在小括号中。

Code**:** e59f303c e3a01001 e5830000 e3a00000 **(**e5811000**)**

### 2.2 根据OOPS的信息推断错误位置

有了这些信息，我们来分析下如何定位到发生错误的代码。总结上面的关键信息。  
出错的位置：first\_drv\_init函数偏移0x94处  
出错的地址：bf000154  
出错的机器码：e5811000

反汇编对应的ko文件。

arm**-**linux**-**objdump **-**D led\_drv**.**ko **>** led\_drv**.**dis

找到init\_module部分的反汇编代码，init\_module这里为first\_drv\_init。找到偏移0x94的地方，也就是154处，从str r1，[r1]可以推断出对应的C代码为\*p = 1。

000000c0 <init\_module>:

c0: e1a0c00d mov ip, sp

148: e3a01001 mov r1, #1 ; 0x1

14c: e5830000 str r0, [r3]

150: e3a00000 mov r0, #0 ; 0x0

154: e5811000 str r1, [r1]

158: e89da818 ldmia sp, {r3, r4, fp, sp, pc}

bf000154这个地址是怎么回事呢？  
查询/proc/kallsyms可以得到first\_drv\_init符号的地址为bf0000c0，偏移0x94，也就是bf000154。

# cat /proc/kallsyms | grep first\_drv\_init

bf0000c0 t first\_drv\_init **[**led\_drv**]**

如果根据汇编代码不好推断呢，可以使用如下命令得到汇编和C语言对应的反汇编文件。

arm**-**linux**-**objdump **-**S led\_drv**.**ko **>** led\_drv**.**dis

从下面的反汇编文件可以看到154对应的C代码内容。

\*p = 1;

return 0;

}

150: e3a00000 mov r0, #0 ; 0x0

154: e5811000 str r1, [r1]

158: e89da818 ldmia sp, {r3, r4, fp, sp, pc}

如果发生oops的内容在内核中，而不是在module中，如何查找到对应的内容呢？  
反汇编vmlinux。

arm**-**linux**-**objdump **-**D vmlinux **>** vmlinux**.**dis

如果想要获取C和汇编的对应关系，需要在生成vmlinux的时候增加-g选项，并在objdump的时候使用-S选项。  
修改源代码的顶层 Makefile增加-g选项。

   CC **=**$**(**CROSSCOM\_PILE**)**gcc            **-->**

   CC **=**$**(**CROSSCOM\_PILE**)**gcc   **-**g

## 3. dump\_stack

该函数会打印寄存器上下文和函数的跟踪线索。

打印的内容如下：

**[<**c002fde8**>]** **(**dump\_stack**+**0x0**/**0x14**)** from **[<**bf000150**>]** **(**first\_drv\_init**+**0x90**/**0xc8 **[**led\_drv**])**

**[<**bf0000c0**>]** **(**first\_drv\_init**+**0x0**/**0xc8 **[**led\_drv**])** from **[<**c006385c**>]** **(**sys\_init\_module**+**0x1424**/**0x1514**)**

r4**:**00000000

**[<**c0062438**>]** **(**sys\_init\_module**+**0x0**/**0x1514**)** from **[<**c002bea0**>]** **(**ret\_fast\_syscall**+**0x0**/**0x2c**)**

## 4. BUG和BUG\_ON()

当这两个宏调用后，会引发OOPS，导致栈的回溯和错误消息的打印。一般把这两个宏当断言使用，如BUG\_ON(bad\_thing)。

## 5. dmesg

dmesg命令用于打印和控制内核环缓冲区。如果开机启动信息没有看到，可以使用dmesg。