目录

一、前言 (1)

1.1 简介 (1)

1.2 问一问 (1)

二、U-Boot设置bootargs参数 (2)

三、Linux内核解析bootargs参数 (4)

3.1 Linux内核解析U-Boot传入的bootargs参数 (4)

3.1.1 保存bootargs (4)

3.1.2 初步解析bootargs (5)

3.1.3 详细解析bootargs之early为1部分 (8)

3.1.4 详细解析bootargs之early为0部分 (13)

3.2 bootargs参数解析扩展 (17)

3.2.1 \*(.init.setup)段 (17)

3.2.2 \*(\_\_param)段 (18)

3.2.3 图述bootargs (19)

3.2 Linux内核解析自己定义的bootargs参数 (19)

3.2.1 Linux内核如何使用自己定义的bootargs参数 (19)

3.2.2 内核启动参数和U-Boot的bootargs参数谁优先? (21)

四、常见bootargs参数解析 (23)

4.1 root (23)

4.2 rootfstype (23)

4.3 console (23)

4.4 mem (23)

4.5 ramdisk\_size (23)

4.6 initrd, noinitrd (24)

4.7 init (24)

4.8 ip (24)

一、前言

1.1 简介

bootargs是环境变量中的重中之重,甚至可以说整个环境变量都是围绕着bootargs来设置的。bootargs的种类非常的多,我们平常只是使用了几种而已。bootargs非常的灵活,内核和文件系统的不同搭配就会有不同的设置方法,甚至你可以不设置bootargs,而直接将其写到内核中去。正是因为这些原因导致了bootargs使用上的困难。

本文则根据bootargs的功能,从U-Boot设置bootargs参数,到Linux内核解析bootargs 参数的整个过程进行了详细的讲解,最后又详细介绍了常见的bootargs参数组合。带领读者从理解bootargs参数原理开始,到最终熟练的使用bootargs参数。

相关联的参考文档:

《张栖银详谈Linux系统启动流程》.pdf

《张栖银详谈Linux标记列表ATAG》.pdf

1.2 问一问

阅读完本文,读者自行即可回答如下问题:

U-Boot如何设置和传递bootargs参数?

bootargs命令行参数的最大长度是多少?

bootargs参数的格式是怎样的,为什么?

U-Boot传入bootargs之后,都去向了哪些地方?

如何配置Linux内核自己的启动参数?

Linux内核分几步解析bootargs,每步重点都做了什么?

为什么说bootargs是所有参数的重中之重,在实际应用中,我们要用哪些参数?

为什么讲内核是先解析early为1的部分,然后再解析early为0的部分?

Linux内核启动的时候是使用配置内核定义的参数,还是使用U-Boot传入的参数?

二、U-Boot设置bootargs参数

在前面一小节《张栖银详谈Linux标记列表ATAG》中,我们知道U-Boot向Linux内核传递参数是单向的,且传递方法是:U-Boot和Linux内核约定采用固定的格式将参数信息存放到固定的内存地址中后,再启动内核,内核启动后从这个地方获得参数。

同时,我们还知道,这个固定的格式就是ATAG标记,如图1所示。

图1 tag标记列表

进入U-Boot-2009.11源码里面的lib\_arm/bootm.c中,找到setup\_commandline\_tag()函数:

char \*commandline = getenv ("bootargs");

setup\_commandline\_tag (bd, commandline);

static void setup\_commandline\_tag (bd\_t \*bd, char \*commandline)

{

char \*p;

if (!commandline)

return;

/\* eat leading white space \*/

for (p = commandline; \*p == ' '; p++);

/\* skip non-existent command lines so the kernel will still use its default command line. \*/ if (\*p == '\0')

return;

params->hdr.tag = ATAG\_CMDLINE;

params->hdr.size = (sizeof (struct tag\_header) + strlen (p) + 1 + 4) >> 2;

strcpy (params->u.cmdline.cmdline, p);

params = tag\_next (params);

}

setup\_commandline\_tag()函数首先对bootargs的有效性进行判断:为空或全部为空字符则不保存bootargs参数;然后将bootargs参数保存为ATAG\_CMDLINE值的ATAG标记。上面的代码中已经有非常完整的解释了。

这里重点讲一下bootargs参数的数据长度的计算方法。

首先,我们来看一看tag\_cmdline结构体的定义:

struct tag\_cmdline {

char cmdline[1]; /\* this is the minimum size \*/

};

这里只是定义了一个只包含1个字节的cmdline数组,实际上这里的目的只是定义一个cmdline指针,因为毕竟具体命令字的长度是根据具体需求而定的。但是,详细阅读过U-Boot 源码之ATAG的读者知道:

#define tag\_size(type) ((sizeof(struct tag\_header) + sizeof(struct type)) >> 2)

所有的ATAG标记列表都是字对齐方式进行保存的。那么对于字符串的bootargs数据来说也不例外。对于字对齐来说,我们重点看低2 bit的数据内容即可,其取值可能为:

0 0 :刚好字对齐;对齐方法可保持不变;

0 1 :多出1个字节;对齐方法要向上增加3个字节;

1 0 :多出2个字节;对齐方法要向上增加2个字节;

1 1 :多出3个字节;对齐方法要向上增加1个字节;

为了保持算法的一致性,统一增加4个字节的方式进行保存,也即向上进1个字大小,然后将低2bit清零,这样就能完整保存bootargs的数据了。但是strlen()只能计算出除‘\0’之外的字符长度,因此在真正的字符串长度是:strlen(p)+1。因此整体的bootargs的字长为: (sizeof(struct tag\_header) //ATAG头的长度

+ (strlen(p) + 1)) //字符串的长度

>>2 //清除第2位

这里对bootargs的设置做个总结:bootargs参数是ATAG标记列表中的一种,为ATAG\_CMDLINE标记,设置bootargs的过程,就是将bootargs字符串保存到ATAG\_CMDLINE 标记列表的过程,并且在保存的过程中,吃掉了开始多余的空格;另外一个重点就是计算出ATAG\_CMDLINE标记实际占用的字长度。

三、Linux内核解析bootargs参数

到此,我们已经知道bootargs参数知识ATAG标记列表中的一种,为ATAG\_CMDLINE标记。那么bootargs在Linux内核的解析,也将遵循ATAG列表解析的流程。详细的内容,可见《张栖银详谈Linux标记列表ATAG》.pdf文档之第三章。下面讲解的内容,源码使用的是Linux-2.6.32.2。

3.1 Linux内核解析U-Boot传入的bootargs参数

整体来讲,Linux内核解析bootargs参数的过程整体分为三个步骤:保存bootargs、初步解析和详细解析。

3.1.1 保存bootargs

在ATAG标记列表解析一文中,我们详细了解到,Linux内核为ATAG\_CMDLINE标记定义了处理函数,如图2所示,并将这些函数相关的struct tagtable结构体存放在了\*(.taglist.init)段中。

图2 Linux内核定义的ATAG解析函数

我们进入parse\_tag\_cmdline()函数,得到如下代码:

static char default\_command\_line[COMMAND\_LINE\_SIZE] \_\_initdata = CONFIG\_CMDLINE; static int \_\_init parse\_tag\_cmdline(const struct tag \*tag)

{

strlcpy(default\_command\_line, tag->u.cmdline.cmdline, COMMAND\_LINE\_SIZE);

return 0;

}

从而可知,该函数的作用就是将U-Boot传入的bootargs参数保存到default\_command\_line全局数组当中,且该数组长度受COMMAND\_LINE\_SIZE的限制(一般为512或者1024)。而且,这里可以看出,不管U-Boot中传入的bootargs参数有多长,该函数始终拷贝的COMMAND\_LINE\_SIZE个数据内容,是不是还有可以优化的地方呢?

这里再贴出parse\_tag\_cmdline()函数的调用过程,详见《张栖银详谈Linux标记列表ATAG》.pdf。

start\_kernel()

—>setup\_arch(&command\_line)

tags = phy\_to\_virt(\_\_atags\_pointer)

if (tags->hdr.tag == ATAG\_CORE) {

if (meminfo.nr\_banks != 0)

squash\_mem\_tags(tags);

save\_atags(tags);

parse\_tags(tags); /\* 解释每个tag \*/

}

—> parse\_tag()

—> t->parse(tag)

3.1.2 初步解析bootargs

经过前面代码对ATAG命令的解析,将bootargs参数保存到了default\_command\_line全局变量中,而且保存的方法是只拷贝COMMAND\_LINE\_SIZE个字节大小。那么就存在两个问题:

1.bootargs命令不长(不足COMMAND\_LINE\_SIZE),default\_command\_line中保存了其他的一些ATAG参数,或者未知的一些数据(RAM的初始数据);

2.bootargs命令过长(超过COMMAND\_LINE\_SIZE),default\_command\_line中保存了部分的bootargs参数,且不是以‘\0’结束。

那么接着setup\_arch()的代码分析,注意各个参数的传递:

void \_\_init setup\_arch(char \*\*cmdline\_p)

{

. . . . . .

char \*from = default\_command\_line;

. . . . . .

if (\_\_atags\_pointer)

tags = phys\_to\_virt(\_\_atags\_pointer);

else if (mdesc->boot\_params)

tags = phys\_to\_virt(mdesc->boot\_params);

. . . . . . .

if (tags->hdr.tag == ATAG\_CORE) {

if (meminfo.nr\_banks != 0)

squash\_mem\_tags(tags);

save\_atags(tags);

parse\_tags(tags);

}

. . . . . .

memcpy(boot\_command\_line, from, COMMAND\_LINE\_SIZE);

boot\_command\_line[COMMAND\_LINE\_SIZE-1] = '\0';

parse\_cmdline(cmdline\_p, from);

. . . . . .

}

注意上面红色标重的代码,前面的内容已经详细讲解过,这里主要注重最后三条语句: 第一条语句将default\_command\_line中的所有数据全部保存进boot\_command\_line全局变量中(from指向default\_command\_line);

第二条语句将boot\_command\_line的最后一个字节赋值为’\0’,那么如果后面的解析bootargs参数时使用boot\_command\_line就解决了前面提的第2个问题了。

第三条语句就是真正的初步解析bootargs的实现了。

parse\_cmdline()函数的实现与parse\_tags()类似,它从".early\_param.init"段中读出struct early\_params结构体数据,并将其中的arg与bootargs参数中的值进行比较,匹配则调用struct

early\_params结构体中的fn()成员函数。

static void \_\_init parse\_cmdline(char \*\*cmdline\_p, char \*from)

{

char c = ' ', \*to = command\_line;

int len = 0;

for (;;) {

if (c == ' ') {

extern struct early\_params \_\_early\_begin, \_\_early\_end;

struct early\_params \*p;

for (p = &\_\_early\_begin; p < &\_\_early\_end; p++) {

int arglen = strlen(p->arg);

if (memcmp(from, p->arg, arglen) == 0) {

if (to != command\_line)

to -= 1;

from += arglen;

p->fn(&from);

/\* 吃掉后面的空格\*/

while (\*from != ' ' && \*from != '\0')

from++;

break;

}

}

}

c = \*from++;

if (!c)

break;

if (COMMAND\_LINE\_SIZE <= ++len)

break;

\*to++ = c;

}

\*to = '\0';

\*cmdline\_p = command\_line;

}

其中的\_\_early\_begin和\_\_early\_end分别表示”.early\_param.init”段的起始地址和结束地址:

\_\_early\_begin = .;

\*(.early\_param.init)

\_\_early\_end = .;

里面一个for循环保准每次遍历”.early\_param.init”段,外面的for死循环保证完全解析bootargs参数(遇到NULL或者解析了COMMAND\_LINE\_SIZE个字节)。

感兴趣的童鞋,可以好好阅读一下本函数的代码,其处理过程还是非常有趣的,该函数

重点做了如下几个事情:

从bootargs参数字符串中一个一个参数的取出来与”.early\_param.init”段进行比较,然后调用对应的p->fn(&from)函数;

从bootargs参数中去除已经被处理过的数据参数内容,并将剩下的数据保存进command\_line静态全局数组中;

让cmdline\_p指向command\_line,这样就让start\_kernel()函数的局部指针变量command\_line指向了setup.c内部的处理后的静态全局变量command\_line数组。

可以在Linux内核中查询\_\_early\_para关键字,可以找到当前内核为bootargs的参数定义了多少处理函数,如图3和图4所示。

图3 内核定义的early\_params结构体

图4 内核为bootargs定义的参数处理函数

可见,bootargs的前期解析,主要针对的是:initrd、nocache、mem等参数,而其他的一些参数放在了后面的详细解析中。这部分先对initrd和mem进行分析是因为接下来就需要建立页表了,这也是为什么输入early\_param的参数很少的原因。

至此,又回到了start\_kernel()函数了,进入详细解析bootargs参数的代码分析了,接着往setup\_arch()函数继续分析,其部分源码如下:

asmlinkage void \_\_init start\_kernel(void)

{

char \* command\_line;

. . . . . .

setup\_arch(&command\_line);

mm\_init\_owner(&init\_mm, &init\_task);

setup\_command\_line(command\_line);

. . . . . .

parse\_early\_param();

parse\_args("Booting kernel", static\_command\_line, \_\_start\_\_\_param,

\_\_stop\_\_\_param - \_\_start\_\_\_param,

&unknown\_bootoption);

}

setup\_arch()函数,前面已经详细介绍过了,接下来的setup\_command\_line()函数主要功能就是对前面提到的bootargs的各种副本分配内存,并进行保存:

static void \_\_init setup\_command\_line(char \*command\_line)

{

saved\_command\_line = alloc\_bootmem(strlen (boot\_command\_line)+1);

static\_command\_line = alloc\_bootmem(strlen (command\_line)+1);

strcpy (saved\_command\_line, boot\_command\_line);

strcpy (static\_command\_line, command\_line);

}

最终,调用了parse\_early\_param(),该函数实现了bootargs参数的一部分解析,也是bootargs解析的重头戏的一部分:解析struct obs\_kernel\_param结构体中early为1的部分;然后直接调用parse\_args()解析struct obs\_kernel\_param结构体中early为0的部分。至此, bootargs参数全部解析完毕。

3.1.3 详细解析bootargs之early为1部分

在对bootargs初步分析和分类型保存之后,下面进入了bootargs解析的重点内容了。首先,我们来看看parse\_early\_options()函数的实现:

void \_\_init parse\_early\_options(char \*cmdline)

{

parse\_args("early options", cmdline, NULL, 0, do\_early\_param);

}

void \_\_init parse\_early\_param(void)

{

static \_\_initdata int done = 0;

static \_\_initdata char tmp\_cmdline[COMMAND\_LINE\_SIZE];

if (done)

return;

/\* All fall through to do\_early\_param. \*/

strlcpy(tmp\_cmdline, boot\_command\_line, COMMAND\_LINE\_SIZE);

parse\_early\_options(tmp\_cmdline);

done = 1;

}

该函数通过静态局部变量done保证只执行一次,然后调用parse\_early\_options()函数对boot\_command\_line进行详细解析。而parse\_early\_options() 最终又调用parse\_args()函数,解析boot\_command\_line命令行参数。细心的朋友可能已经发现了,parse\_args()正是详细解析bootargs之early为0部分解析时调用的那个函数,只是具体传入的参数不同而已,这里首先将其列出,以便读者对比:

parse\_args("early options", cmdline, NULL, 0, do\_early\_param);

parse\_args("Booting kernel", static\_command\_line, \_\_start\_\_\_param,

本文档下载自文档之家，www.doczj.com-免费文档分享平台，众多试卷、习题答案、公务员考试、英语学习、法语学习、人力资源管理、电脑基础知识、学习计划、工作计划、工作总结、活动策划、企业管理等文档分类免费下载；乐于分享，共同进步，转载请保留出处:http://www.doczj.com/doc/25dfce8e336c1eb91b375d7d.html