uboot的relocation原理详细分析

- 编辑
- 删除

最近在一直在做uboot的移植工作，uboot中有很多值得学习的东西，之前总结过uboot的启动流程，但uboot一个非常核心的功能没有仔细研究，就是uboot的relocation功能。

这几天研究下uboot的relocation功能，记录在此，跟大家共享。

自己辛苦编辑，转载请注明出处，谢谢！

所谓的relocation，就是重定位，uboot运行后会将自身代码拷贝到sdram的另一个位置继续运行，这个在uboot启动流程分析中说过。

但基于以前的理解，一个完整可运行的bin文件，link时指定的链接地址，load时的加载地址，运行时的运行地址，这3个地址应该是一致的

relocation后运行地址不同于加载地址 特别是链接地址，ARM的寻址会不会出现问题？

新版uboot跟老版uboot不太一样的地方在于新版uboot不管uboot的load addr（entry pointer）在哪里，启动后会计算出一个靠近sdram顶端的地址，将自身代码拷贝到该地址，继续运行。

个人感觉uboot这样改进用意有二，一是为kernel腾出低端空间，防止kernel解压覆盖uboot，二是对于由静态存储器（spiflash nandflash）启动，这个relocation是必须的。

但是这样会有一个问题，relocation后uboot的运行地址跟其链接地址不一致，compiler会在link时确定了其中变量以及函数的绝对地址，链接地址 加载地址 运行地址应该一致，

这样看来，arm在寻址这些变量 函数时找到的应该是relocation之前的地址，这样relocation就没有意义了！

当然uboot不会这样，我们来分析一下uboot下relocation之后是如何寻址的，开始学习之前我是有3个疑问，如下

（1）如何对函数进行寻址调用

（2）如何对全局变量进行寻址操作（读写）

（3）对于全局指针变量中存储的其他变量或函数地址在relocation之后如何操作

搞清楚这3个问题，对于我来说relocation的原理就算是搞明白了。

为了搞清楚这些，在uboot的某一个文件中加入如下代码

[cpp] view plain copy print?

```
1. void test_func(void)
2. {
3.     printf("test func\n");
4. }
5.
6. static void * test_func_val = test_func;
7. static int test_val = 10;
8.
9. void rel_dyn_test()
10. {
11.     test_val = 20;
12.     printf("test = 0x%x\n", test_func);
13.     printf("test_func = 0x%x\n", test_func_val);
14.     test_func();
15. }
```

```
void test_func(void)
{
    printf("test func\n");
}

static void * test_func_val = test_func;
static int test_val = 10;

void rel_dyn_test()
{
    test_val = 20;
    printf("test = 0x%x\n", test_func);
    printf("test_func = 0x%x\n", test_func_val);
    test_func();
}
```

rel_dyn_test函数中就包含了函数指针 变量赋值 函数调用这3种情况，寻址肯定要汇编级的追踪才可以，编译完成后反汇编，得到u-boot.dump（objdump用-D选项，将所有section都disassemble出来）

找到rel_dyn_test函数，如下：

[cpp] view plain copy print?

```
1. 80e9d3cc <test_func>:
2. 80e9d3cc:   e59f0000    ldr r0, [pc, #0]    ; 80e9d3d4 <test_func+0x8>
3. 80e9d3d0:   eaffc2fb    b   80e8dfc4 <printf>
4. 80e9d3d4:   80eb1c39    .word   0x80eb1c39
5.
6. 80e9d3d8 <rel_dyn_test>:
7. 80e9d3d8:   e59f202c    ldr r2, [pc, #44]   ; 80e9d40c <rel_dyn_test+0x34>
8. 80e9d3dc:   e3a03014    mov r3, #20 ; 0x14
9. 80e9d3e0:   e92d4010    push    {r4, lr}
10. 80e9d3e4:   e59f1024    ldr r1, [pc, #36]   ; 80e9d410 <rel_dyn_test+0x38>
11. 80e9d3e8:   e5823000    str r3, [r2]
12. 80e9d3ec:   e59f0020    ldr r0, [pc, #32]   ; 80e9d414 <rel_dyn_test+0x3c>
13. 80e9d3f0:   ebffc2f3    bl  80e8dfc4 <printf>
14. 80e9d3f4:   e59f301c    ldr r3, [pc, #28]   ; 80e9d418 <rel_dyn_test+0x40>
15. 80e9d3f8:   e59f001c    ldr r0, [pc, #28]   ; 80e9d41c <rel_dyn_test+0x44>
16. 80e9d3fc:   e5931000    ldr r1, [r3]
17. 80e9d400:   ebffc2ef    bl  80e8dfc4 <printf>
18. 80e9d404:   e8bd4010    pop {r4, lr}
19. 80e9d408:   eafffffef   b   80e9d3cc <test_func>
20. 80e9d40c:   80eb75c0    .word   0x80eb75c0
21. 80e9d410:   80e9d3cc    .word   0x80e9d3cc
22. 80e9d414:   80eb1c44    .word   0x80eb1c44
23. 80e9d418:   80eaa54c    .word   0x80eaa54c
24. 80e9d41c:   80eb1c51    .word   0x80eb1c51
```

```
80e9d3cc <test_func>:
80e9d3cc:    e59f0000    ldr r0, [pc, #0]    ; 80e9d3d4 <test_func+0x8>
80e9d3d0:    eaffc2fb    b   80e8dfc4 <printf>
80e9d3d4:    80eb1c39    .word   0x80eb1c39

80e9d3d8 <rel_dyn_test>:
80e9d3d8:    e59f202c    ldr r2, [pc, #44]   ; 80e9d40c <rel_dyn_test+0x34>
80e9d3dc:    e3a03014    mov r3, #20 ; 0x14
```

```
80e9d3e0:   e92d4010    push    {r4, lr}
80e9d3e4:   e59f1024    ldr r1, [pc, #36]   ; 80e9d410 <rel_dyn_test+0x38>
80e9d3e8:   e5823000    str r3, [r2]
80e9d3ec:   e59f0020    ldr r0, [pc, #32]   ; 80e9d414 <rel_dyn_test+0x3c>
80e9d3f0:   ebffc2f3    bl  80e8dfc4 <printf>
80e9d3f4:   e59f301c    ldr r3, [pc, #28]   ; 80e9d418 <rel_dyn_test+0x40>
80e9d3f8:   e59f001c    ldr r0, [pc, #28]   ; 80e9d41c <rel_dyn_test+0x44>
80e9d3fc:   e5931000    ldr r1, [r3]
80e9d400:   ebffc2ef    bl  80e8dfc4 <printf>
80e9d404:   e8bd4010    pop {r4, lr}
80e9d408:   eaffffef    b   80e9d3cc <test_func>
80e9d40c:   80eb75c0    .word   0x80eb75c0
80e9d410:   80e9d3cc    .word   0x80e9d3cc
80e9d414:   80eb1c44    .word   0x80eb1c44
80e9d418:   80eaa54c    .word   0x80eaa54c
80e9d41c:   80eb1c51    .word   0x80eb1c51
```

。。。

data段中

[cpp] view plain copy print?

```
1. 80eb75c0 <test_val>:
2. 80eb75c0:   0000000a    .word   0x0000000a
```

```
80eb75c0 <test_val>:
80eb75c0:   0000000a    .word   0x0000000a
```

。。。

[cpp] view plain copy print?

```
1. 80eaa54c <test_func_val>:
2. 80eaa54c:   80e9d3cc    .word   0x80e9d3cc
```

```
80eaa54c <test_func_val>:
80eaa54c:   80e9d3cc    .word   0x80e9d3cc
```

rel_dyn_test反汇编后，最后多了一部分从0x80e9d40c开始的内存空间，对比发现这部分内存空间地址上的值竟然是函数需要的变量test_val test_func_val的地址。

网上资料称这些函数末尾存储变量地址的内存空间为Label，（编译器自动分配）

一条条指令来分析。

ldr r2, [pc, #44] ========> r2 = [pc + 0x2c]========>r2 = [0x80e9d3e0 + 0x2c]========>r2 =[0x80e9d40c]

需要注意，由于ARM的流水线机制，当前PC值为当前地址加8个字节

这样r2获取的是0x80e9d40c地址的值0x80eb75c0，这就是test_val的值嘛

mov r3， #20=======> r3 = 20

对应C函数这应该是为test_val = 20做准备，先跳过后面2条指令，发现

str r3, [r2]

很明显了，将立即数20存入0x80eb75c0中也就是test_val中。

这3条指令说明，ARM对于变量test_val的寻址如下：

（1）将变量test_val的地址存储在函数尾端的Label中（这段内存空间是由编译器自动分配的，而非人为）

（2）基于PC相对寻址获取函数尾端Label上的变量地址

（3）对test_val变量地址进行读写操作

再来看其中的几条指令

ldr r3, [pc, #28] ======> r3 = [0x80e9d3fc + 0x1c] ======> r3 = [0x80e9d418] =====> r3 = 0x80eaa54c

ldr r1, [r3] ======> r1 = [0x80eaa54c] =======> r1 = 0x80e9d3cc

0x80e9d3cc这个地址可以看出是test_func的入口地址，这里是printf打印test_func_val的值

可以看出对于函数指针变量的寻址跟普通变量一样。

接下来来看函数的调用，可以看到对于printf以及test_func，使用的是指令bl以及b进行跳转，这2条指令都是相对寻址（pc + offset）

说明ARM调用函数使用的是相对寻址指令bl或b，与函数的绝对地址无关

对于这3种情况的寻址方法已经知道了，那就需要思考一下relocation之后会有什么变化。

将rel_dyn_test relocation之后可以想象，函数的调用还是没有问题的，因为使用了bl或b相对跳转指令。

但是对于变量的寻址就有问题了，寻址的前2步没有问题，**相对寻址获取尾部Label中的变量地址，但获取的变量地址是在 link时就确定下来的绝对地址啊！**

而对于指针变量的寻址呢，问题更多了，

首先跟普通变量寻址一样，尾部内存空间的变量地址是link时的绝对地址，再者，指针变量存储的变量指针或者函数指针也是在link时确定的绝对地址，relocation之后这个值也变了！

那uboot是如何来处理这些情况的呢？更准确的说应该是compiler和uboot如何一起来处理这些情况的呢？

这里利用了PIC位置无关代码，通过为编译器指定编译选项-fpic或-fpie产生，这样编译产生的目标文件包含了PIC所需要的信息，-fpic，-fpie是gcc的PIC编译选项。ld也有PIC连接选项-pie，要获得一个完整的PIC可运行文件，连接目标文件时必须为ld指定-pie选项，

察看uboot的编译选项发现，在arch/arm/config.mk，如下：

[cpp] view plain copy print?

```
1. # needed for relocation
2. LDFLAGS_u-boot += -pie
```

```
# needed for relocation
LDFLAGS_u-boot += -pie
```

uboot只指定了-pie给ld，而没有指定-fPIC或-fPIE给gcc。

指定-pie后编译生成的uboot中就会有一个rel.dyn段，**uboot就是靠rel.dyn段实现了完美的relocation！**

察看u-boot.dump中的rel.dyn段，如下：

[cpp] view plain copy print?

```
1. Disassembly of section .rel.dyn:
2.
3. 80eb7d54 <__rel_dyn_end-0x5c10>:
4. 80eb7d54:	80e80020	rschi	r0, r8, r0, lsr #32
5. 80eb7d58:	00000017	andeq	r0, r0, r7, lsl r0
6. 80eb7d5c:	80e80024	rschi	r0, r8, r4, lsr #32
7. 80eb7d60:	00000017	andeq	r0, r0, r7, lsl r0
8. 80eb7d64:	80e80028	rschi	r0, r8, r8, lsr #32
9. 80eb7d68:	00000017	andeq	r0, r0, r7, lsl r0
10. 。。。
```

```
11. <span style="color:#FF0000;">80eba944:      80e9d40c       rschi   sp, r9, ip, lsl #8
12. 80eba948:      00000017       andeq   r0, r0, r7, lsl r0
13. 80eba94c:      80e9d410       rschi   sp, r9, r0, lsl r4
14. 80eba950:      00000017       andeq   r0, r0, r7, lsl r0
15. 80eba954:      80e9d414       rschi   sp, r9, r4, lsl r4
16. 80eba958:      00000017       andeq   r0, r0, r7, lsl r0
17. 80eba95c:      80e9d418       rschi   sp, r9, r8, lsl r4
18. 80eba960:      00000017       andeq   r0, r0, r7, lsl r0
19. 80eba964:      80e9d41c       rschi   sp, r9, ip, lsl r4
20. 80eba968:      00000017       andeq   r0, r0, r7, lsl r0</span>
21. 。。。。
```

```
Disassembly of section .rel.dyn:

80eb7d54 <__rel_dyn_end-0x5c10>:
80eb7d54:       80e80020        rschi   r0, r8, r0, lsr #32
80eb7d58:       00000017        andeq   r0, r0, r7, lsl r0
80eb7d5c:       80e80024        rschi   r0, r8, r4, lsr #32
80eb7d60:       00000017        andeq   r0, r0, r7, lsl r0
80eb7d64:       80e80028        rschi   r0, r8, r8, lsr #32
80eb7d68:       00000017        andeq   r0, r0, r7, lsl r0
。。。
<span style="color:#FF0000;">80eba944:       80e9d40c        rschi   sp, r9, ip, lsl
#8
80eba948:       00000017        andeq   r0, r0, r7, lsl r0
80eba94c:       80e9d410        rschi   sp, r9, r0, lsl r4
80eba950:       00000017        andeq   r0, r0, r7, lsl r0
80eba954:       80e9d414        rschi   sp, r9, r4, lsl r4
80eba958:       00000017        andeq   r0, r0, r7, lsl r0
80eba95c:       80e9d418        rschi   sp, r9, r8, lsl r4
80eba960:       00000017        andeq   r0, r0, r7, lsl r0
80eba964:       80e9d41c        rschi   sp, r9, ip, lsl r4
80eba968:       00000017        andeq   r0, r0, r7, lsl r0</span>
。。。。
```

有没有注意到，rel_dyn_test末尾存储全局变量地址的Label地址也存储在这里，那有什么用呢，**那就来看一下uboot的核心函数relocate_code是如何实现自身的relocation的**，

在arch/arm/lib/relocate.S中

[cpp] view plain copy print?

```
1. ENTRY(relocate_code)
2.     ldr     r1, =__image_copy_start /* r1 <- SRC &__image_copy_start */
3.     subs    r4, r0, r1              /* r4 <- relocation offset */
4.     beq     relocate_done           /* skip relocation */
5.     ldr     r2, =__image_copy_end   /* r2 <- SRC &__image_copy_end */
6. 
7. copy_loop:
8.     ldmia   r1!, {r10-r11}          /* copy from source address [r1]    */
9.     stmia   r0!, {r10-r11}          /* copy to   target address [r0]    */
10.     cmp     r1, r2                  /* until source end address [r2]    */
11.     blo     copy_loop
12. 
13.     /*
14.      * fix .rel.dyn relocations
15.      */
16.     ldr     r2, =__rel_dyn_start    /* r2 <- SRC &__rel_dyn_start */
17.     ldr     r3, =__rel_dyn_end      /* r3 <- SRC &__rel_dyn_end */
18. fixloop:
19.     ldmia   r2!, {r0-r1}            /* (r0,r1) <- (SRC location,fixup) */
20.     and     r1, r1, #0xff
21.     cmp     r1, #23                 /* relative fixup? */
22.     bne     fixnext
23. 
24.     /* relative fix: increase location by offset */
25.     add     r0, r0, r4
26.     ldr     r1, [r0]
27.     add     r1, r1, r4
28.     str     r1, [r0]
29. fixnext:
30.     cmp     r2, r3
31.     blo     fixloop
32. 
33. relocate_done:
```

```
ENTRY(relocate_code)
        ldr     r1, =__image_copy_start /* r1 <- SRC &__image_copy_start */
        subs    r4, r0, r1              /* r4 <- relocation offset */
```

```
        beq     relocate_done           /* skip relocation */
        ldr     r2, =__image_copy_end   /* r2 <- SRC &__image_copy_end */

copy_loop:
        ldmia   r1!, {r10-r11}          /* copy from source address [r1]    */
        stmia   r0!, {r10-r11}          /* copy to   target address [r0]    */
        cmp     r1, r2                  /* until source end address [r2]    */
        blo     copy_loop

        /*
         * fix .rel.dyn relocations
         */
        ldr     r2, =__rel_dyn_start    /* r2 <- SRC &__rel_dyn_start */
        ldr     r3, =__rel_dyn_end      /* r3 <- SRC &__rel_dyn_end */
fixloop:
        ldmia   r2!, {r0-r1}            /* (r0,r1) <- (SRC location,fixup) */
        and     r1, r1, #0xff
        cmp     r1, #23                 /* relative fixup? */
        bne     fixnext

        /* relative fix: increase location by offset */
        add     r0, r0, r4
        ldr     r1, [r0]
        add     r1, r1, r4
        str     r1, [r0]
fixnext:
        cmp     r2, r3
        blo     fixloop

relocate_done:
```

前半部分在uboot启动流程中讲过，将__image_copy_start到__image_copy_end之间的数据进行拷贝

来看一下arm的link script，在arch/arm/cpu/u-boot.lds，如下：

[cpp] view plain copy print?

```
1. OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
2. OUTPUT_ARCH(arm)
```

```
3. ENTRY(_start)
4. SECTIONS
5. {
6.     . = 0x00000000;
7.
8.     . = ALIGN(4);
9.     .text :
10.     {
11.         *(.__image_copy_start)
12.         CPUDIR/start.o (.text*)
13.         *(.text*)
14.     }
15.
16.     . = ALIGN(4);
17.     .rodata : { *(SORT_BY_ALIGNMENT(SORT_BY_NAME(.rodata*))) }
18.
19.     . = ALIGN(4);
20.     .data : {
21.         *(.data*)
22.     }
23.
24.     . = ALIGN(4);
25.
26.     . = .;
27.
28.     . = ALIGN(4);
29.     .u_boot_list : {
30. <pre name="code" class="cpp">            KEEP(*(SORT(.u_boot_list*)));
31.     }
32.
33.     . = ALIGN(4);
34.
35.     .image_copy_end :
36.     {
37.         *(.__image_copy_end)
38.     }
39.
40.     .rel_dyn_start :
41.     {
42.         *(.__rel_dyn_start)
43.     }
```

```
    .
    .rel.dyn : {
        *(.rel*)
    }
    .
    .rel_dyn_end :
    {
        *(.__rel_dyn_end)
    }
    .
    .end :
    {
        *(.__end)
    }
    .
    _image_binary_end = .;
```

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
        . = 0x00000000;

        . = ALIGN(4);
        .text :
        {
                *(.__image_copy_start)
                CPUDIR/start.o (.text*)
                *(.text*)
        }

        . = ALIGN(4);
        .rodata : { *(SORT_BY_ALIGNMENT(SORT_BY_NAME(.rodata*))) }

        . = ALIGN(4);
        .data : {
                *(.data*)
        }
```

```
        . = ALIGN(4);

        . = .;

        . = ALIGN(4);
        .u_boot_list : {
<pre name="code" class="cpp">                KEEP(*(SORT(.u_boot_list*)));
        }

        . = ALIGN(4);

        .image_copy_end :
        {
                *(.__image_copy_end)
        }

        .rel_dyn_start :
        {
                *(.__rel_dyn_start)
        }

        .rel.dyn : {
                *(.rel*)
        }

        .rel_dyn_end :
        {
                *(.__rel_dyn_end)
        }

        .end :
        {
                *(.__end)
        }

        _image_binary_end = .;
```

可以看出__image_copy_start---end之间包括了text data rodata段，但是没有包括rel_dyn。
继续看relocate_code函数，拷贝__image_copy_start----end之间的数据，但没有拷贝rel.dyn段。
首先获取__rel_dyn_start地址到r2，将start地址上连续2个4字节地址的值存在r0 r1中
判断r1中的值低8位，如果为0x17，则将r0中的值加relocation offset。
获取以此r0中值为地址上的值，存到r1中
将r1中值加relocation offset，再存回以r0中值为地址上。
以此循环，直到__rel_dyn_end。
这样读有些拗口。来以咱们的rel_dyn_test举例子。
上面rel.dyn段中有一段如下：

[cpp] view plain copy print?

```
1. 80eba944:    80e9d40c    rschi   sp, r9, ip, lsl #8
2. 80eba948:    00000017    andeq   r0, r0, r7, lsl r0
```

```
80eba944:    80e9d40c    rschi   sp, r9, ip, lsl #8
80eba948:    00000017    andeq   r0, r0, r7, lsl r0
```

按照上面的分析，判断第二个四字节为0x17，r0中存储为0x80e9d40c。这个是rel_dyn_test末尾Label的地址啊，
将r0加上relocation offset，则到了relocation之后rel_dyn_test的末尾Label。
获取r0为地址上的值到r1中，0x80eb75c0，可以看到，这个值就是变量test_val的首地址啊。
最后将r1加上relocation offset，写回以r0为地址上。意思是将变量test_val地址加offset后写回到relocation之后rel_dyn_test的末尾Label中。
这样relocate_code完成后，再来看对test_val的寻址。寻址第三步获取到的是修改之后的relocation addr啊，这样就可以获取到relocation之后的test_val值！
对于普通变量寻址是这样，那对于指针变量呢，如test_func_val呢？
获取test_func_val relocation后地址的步骤跟上面一样，但是我们在获取test_func_val的值时要注意，这个变量存储的是函数test_func指针，之前是

0x80e9d3cc，relocation之后就变化了，所以test_func_val的值也应该变化，这个该怎么办？

方法是一样的，可以在rel.dyn段中找到如下一段：

[cpp] view plain copy print?

```
1. 80ebc18c:	80eaa54c	rschi	sl, sl, ip, asr #10
2. 80ebc190:	00000017	andeq	r0, r0, r7, lsl r0
```

```
80ebc18c:	80eaa54c	rschi	sl, sl, ip, asr #10
80ebc190:	00000017	andeq	r0, r0, r7, lsl r0
```

这上面存储的是test_func_val的地址，按照relocate_code的操作，完成后80eaa54c + offset上的值也应该+offset了。

这就解决了，test_func_val的值也就是test_func的地址也被修改为relocation之后的地址了。