(1条消息**) u-boot-2021.01**（**imx6ull**）启动流程分析之四：**relocate_code**和**relocate_vectors**重定位分析**___ASDFGH**的博客**-CSDN**博客

3.4.3 relocate_code

顾名思义，代码重定位。

```
/* file: arch/arm/lib/relocate.S */
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, #R_ARM_RELATIVE
        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:
...
        /* ARMv4- don't know bx lr but the assembler fails to see that */

#ifdef __ARM_ARCH_4__
        mov     pc, lr
#else
        bx      lr
#endif

ENDPROC(relocate_code)
```

从上面的代码可以看到，先将__image_copy_start~__image_copy_end部分重定位，根据u-boot.lds文件可以知道，这个区间就已经包含了vendor向量、text代码段和data段等等。

3.4.3.1 “image”部分重定位过程

在进入relocate_code部分之前，r0寄存器就保存了重定位的地址值（这点可以回到_main函数中验证）。

1. 把代码链接地址赋给r1寄存器；
2. 计算r4=r0-r1=“重定位目标地址”-“重定位源地址”，得到的是地址偏移量，**r4**这个偏移值后面在**.rel.dyn**段的配置会用上；
3. 如果r1和r0相等，那就根本不用重定位了，直接跳到重定位完毕的relocate_done标志去了。但是，如果刚才一路追踪着内存地址的设置，就会发现它们不会相等，所以需要重定位；
4. 通过ldmia命令（**ld**r/**m**any/**i**ncrease/**a**fter）读取”重定位源地址r1“开始的2个32位地址的数据保存到r10和r11寄存器中（完成后r1的值会更新）；
5. 然后又将r10和r11两个寄存器的值存放到“重定位目标地址r0”开始的2个地址处（完成后r0的值也会更新），这样就完成了2个32位的代码重定位；
6. 完成后判断“更新后的源地址r1”和“重定位结束的地址r2”是否相等，相等则代表重定位完成，不相等则需要返回第4步继续拷贝。

3.4.3.2 .rel.dyn部分配置过程

在分析.rel.dynd段fix（配置）之前，需要先知道的一点就是：程序运行时应当处于链接地址，要想访问某个变量，就得先得到变量的地址。 但由于重定位代码之后地址会发生改变，当需要访问这些数据时，就不应该使用绝对地址。在u-boot程序中是使用了pc寄存器来偏移获得这个数据的Label值，Label就保存着这个变量的地址，一旦知道Label所保存的地址那就可以访问这个数据，所以重定位之后修改Label所保存的值就能正确地访问了。至于它的原理，举个例子看一下就清楚了，这里就添加一个函数来查看一下变量的存储方式（仅供测试，不考虑程序运行情况）：

```
@@ -951,8 +951,18 @@ static const init_fnc_t init_sequence_f[] = {
        NULL,
 };

+int rel_test_val = 0x100;
+
+void rel_test_fun(void)
+{
+       rel_test_val = 0x200;
+       printf("rel_test!\n");
+}
+
 void board_init_f(ulong boot_flags)
 {
+       rel_test_fun();
+
        gd->flags = boot_flags;
        gd->have_console = 0;
```

再执行`arm-linux-gnueabihf-objdump -D -m arm u-boot > u-boot.dis`进行反汇编一下看看关于这部分的实现：

```
/* file: u-boot.dis */
// Disassembly of section .text_rest:
8780b978 <rel_test_fun>:
8780b978: f44f 7200   mov.w r2, #512  ; 0x200
8780b97c: 4b02        ldr r3, [pc, #8]  ; (8780b988 <rel_test_fun+0x10>)
8780b97e: 4803        ldr r0, [pc, #12] ; (8780b98c <rel_test_fun+0x14>)
8780b980: 601a        str r2, [r3, #0]
8780b982: f025 be68   b.w 87831656 <printf>
8780b986: bf00        nop
8780b988: 878469d0      ; <UNDEFINED> instruction: 0x878469d0
8780b98c: 8783fa93      ; <UNDEFINED> instruction: 0x8783fa93

8780b990 <board_init_f>:
8780b990: b570        push  {r4, r5, r6, lr}
8780b992: 4604        mov r4, r0
8780b994: f7ff fff0   bl  8780b978 <rel_test_fun>
8780b998: 2200        movs  r2, #0
8780b99a: 4d0e        ldr r5, [pc, #56] ; (8780b9d4 <board_init_f+0x44>)
...

// Disassembly of section .data:
878469d0 <rel_test_val>:
878469d0: 00000100    andeq r0, r0, r0, lsl #2
...

// Disassembly of section .rel.dyn:
8784b6dc:   8780b988    strhi   fp, [r0, r8, lsl #19]
8784b6e0:   00000017    andeq   r0, r0, r7, lsl r0
```

看到以上反汇编内容，一条一条指令分析（冒号左侧为内存地址，右侧为机器码、指令）：

- 8780b994：`board_init_f`函数通过bl命令跳转到了`8780b978`地址处的`rel_test_fun`函数；
- 8780b978：在`rel_test_fun`函数中，将0x200写入到r2寄存器；
- 8780b97c：将pc寄存器的值加上8赋给r3寄存器。由于ARM三级流水线结构（在执行第一条指令时，第二条指令正在译码阶段，第三条指令正在取指阶段，pc寄存器就是指向取指的地址。参考文章），而随后的2条指令都是16位的thumb指令集，所以pc=8780b97c+2+2=8780b980，而不是8780b97c+4+4，故r3=0x8780b980+8=0x8780b988；

- 8780b988：它随着程序地址变化而变化，俗称位置无关码，也就是**“Label”**，它保存的就是一个地址。在这个例子中保存rel_test_val的地址值是878469d0；
- 878469d0：rel_test_val的地址，冒号右侧就是保存着它的值；
- 8780b980：将r2的0x200保存到r3保存的地址中，这样就完成了一次rel_test_val的赋值。

这个函数在重定位之前是正常运行的，但是重定位之后，代码和数据的地址都发生了变化，而Label处保存的值还是原样的，所以访问该数据就还是重定位之前的数据。

然而，在编译u-boot的时候已经加上了**“-pie”**选项，它的作用就是生成位置无关码，编译时生成一个.rel.dyn段，通过这个段可以对重定位后的代码进行“补充纠正”。那就是`relocate_code`函数后半部分的**“fix配置”**了，它的过程如下：

1. 首先使用ldmia命令读取r2保存的地址开始的2个32位地址的值存到r0和r1（r1保存的是高地址的值）；
2. 将r1寄存器和0xff进行**“&运算”**后保存到r1寄存器，目的是想获得低8位的值；
3. 将r1和R_ARM_RELATIVE比较，这个值在`include/elf.h`文件中定义，它的值为23，也就是16进制的0x17；
4. 如果r1的低8位不是0x17，则跳到后面比较地址判断.rel.dyn段是否已经配置完成，如果地址不相等则返回去继续第1步；
5. 如果r1的低8位是0x17，则代表r0保存的地址值是一个**“Label”**，例如上面反汇编文件中的8780b988地址；
6. 更新**“Label”**保存的值：

   ① 将Label保存的地址加上重定位偏移的值r4得到新Label，假设r4=0x10002000，那么就可以知道重定位后新Label是在8780b988+10002000=9780d988处，把它保存到r0寄存器中；

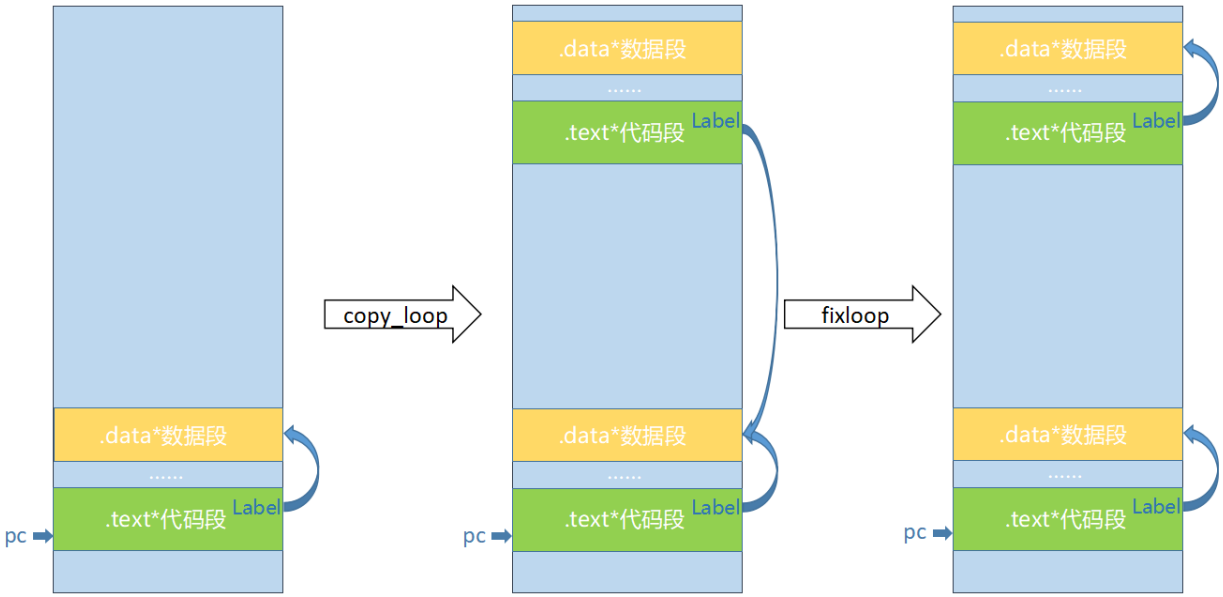
   ② 读取r0（新Label）的值存到r1寄存器中，这时新Label保存的还是变量旧的地址：878469d0；

   ③ 将r1（变量旧的地址）也加上重定位偏移的值r4，就变成了878469d0+10002000=978489d0；

   ④ 再将r1的值（变量新的地址978489d0）写回到r0（新Label）处，这样就完成了一次Label的更新。

其实，以上6个步骤的操作可以总结为一句话：将重定位后代码段中的**Label**所保存的地址值更新为对应的新地址。 以上面反汇编为例，在重定位后将Label保存的地址值878469d0更新为978489d0，这样才能正确地访问到更新后的rel_test_val变量。

整个`relocate_code`代码重定位可以简化为一张图：



relocate_code执行过程

需要注意的是，在`relocate_code`函数末尾部分可以看到以下部分：

```
#ifdef __ARM_ARCH_4__
        mov     pc, lr
#else
        bx      lr
#endif
```

由于没有定义`__ARM_ARCH_4__`，所以执行的是`bx lr`语句。需要知道的一点是，当使用bl或blx跳转去执行的时候，lr（r14）寄存器保存的是pc（r15）寄存器减4的地址，也就是返回的地址（参考文章），所以这时还没有跳转到重定位之后的内存地址中去。那它是什么时候跳转的呢？它又是如何跳转的呢？那就是在`_main`函数里调用完`relocate_code`和其他一些函数之后，才使用绝对跳转跳到重定位之后的地址去的：

```
#if CONFIG_IS_ENABLED(SYS_THUMB_BUILD)
        ldr     lr, =board_init_r       /* this is auto-relocated! */
        bx      lr
#else
        ldr     pc, =board_init_r       /* this is auto-relocated! */
#endif
```

这里知道跳转的时机和方式即可，board_init_r函数后面会研究。

3.4.4 relocate_vectors

从名字也可以知道它的功能，它就是重定位vectors向量表:

```
/* file: arch/arm/lib/relocate.S */
ENTRY(relocate_vectors)

#ifdef CONFIG_CPU_V7M
        /*
         * On ARMv7-M we only have to write the new vector address
         * to VTOR register.
         */
        ldr     r0, [r9, #GD_RELOCADDR] /* r0 = gd->relocaddr */
        ldr     r1, =V7M_SCB_BASE
        str     r0, [r1, V7M_SCB_VTOR]
#else
#ifdef CONFIG_HAS_VBAR
        /*
         * If the ARM processor has the security extensions,
         * use VBAR to relocate the exception vectors.
         */
        ldr     r0, [r9, #GD_RELOCADDR] /* r0 = gd->relocaddr */
        mcr     p15, 0, r0, c12, c0, 0  /* Set VBAR */
#else
        /*
         * Copy the relocated exception vectors to the
         * correct address
         * CP15 c1 V bit gives us the location of the vectors:
         * 0x00000000 or 0xFFFF0000.
         */
        ldr     r0, [r9, #GD_RELOCADDR] /* r0 = gd->relocaddr */
        mrc     p15, 0, r2, c1, c0, 0   /* V bit (bit[13]) in CP15 c1 */
        ands    r2, r2, #(1 << 13)
        ldreq   r1, =0x00000000         /* If V=0 */
        ldrne   r1, =0xFFFF0000         /* If V=1 */
        ldmia   r0!, {r2-r8,r10}
        stmia   r1!, {r2-r8,r10}
        ldmia   r0!, {r2-r8,r10}
        stmia   r1!, {r2-r8,r10}
#endif
#endif
        bx      lr

ENDPROC(relocate_vectors)
```

根据宏定义的条件判断，最终由于判断CONFIG_HAS_VBAR部分条件为真，所以整段代码也就只有简化成以下三句内容:

```
ldr     r0, [r9, #GD_RELOCADDR]
mcr     p15, 0, r0, c12, c0, 0
bx      lr
```

根据注释也可以很清楚知道，它就只是将cp15协处理器对应的VBAR设置为重定位之后的uboot起始地址，也即vectors的地址。因为vectors区是在uboot的最前面，而前面的relocate_code已经也将它一起重定位了，所以只需要简单设置新的地址即可。（还记得前面_start部分已经设置过cp15寄存器支持重定向vectors区并且设置当时的地址，现在重定位之后当然需要将地址修改过来。）

重定位完成之后，就继续调用board_init_r函数初始化。

未完待续...